

# Microsoft. Operating System / 2

---

Windows Presentation Manager  
Reference

Volume 1

Microsoft Corporation

Information in this document is subject to change without notice and does not represent a commitment on the part of Microsoft Corporation. The software described in this document is furnished under a license agreement or nondisclosure agreement. The software may be used or copied only in accordance with the terms of the agreement. It is against the law to copy this software on magnetic tape, disk, or any other medium for any purpose other than the purchaser's personal use.

© Copyright Microsoft Corporation, 1987

Microsoft, the Microsoft logo, MS-DOS, and MS are registered trademarks of Microsoft Corporation.

Document Number 07-01-87-001  
Part Number 00248

# Contents

---

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Application Model</b>	<b>13</b>
<b>3</b>	<b>User Interface</b>	<b>45</b>
<b>4</b>	<b>Window Management Functions</b>	<b>155</b>

# Figures

---

Figure 1.1	Typical Presentation Manager Screen Layout	6
Figure 1.2	Presentation Manager window with frame	8
Figure 1.3	Menu bar with pull-down menu	10
Figure 2.1	Presentation Manager - System Structure.	17
Figure 2.2	Application Model for Input and Output.	21
Figure 2.4	Presentation Manager application model for dialog boxes	29
Figure 2.5	Application Structure.	32
Figure 3.1	The File Cabinet with Tree	64
Figure 3.2	The File pull-down	66
Figure 3.3	Key/Mouse Click Usages for Selection and Manipulation	70
Figure 3.4	Key/Mouse Drag Usages for Selection and Manipulation	70
Figure 3.5	The Filing system with Options pull-down	71
Figure 3.6	The Filing system with Special menu	73
Figure 3.7	The File Cabinet with Window pull-down	74
Figure 3.8	The File Cabinet with STARTUP panel	76
Figure 3.9	Startup Editor - main panel	78
Figure 3.10	Startup Editor - File pull down	79
Figure 3.11	Startup Editor - Edit pull down	80
Figure 3.12	The Task Manager window	82
Figure 3.13	The Task Manager window with Control pull down	83
Figure 3.14	Task Manager - terminating a task	84
Figure 3.15	Task Manager with Shutdown pull down	85
Figure 3.16	Control Panel	86
Figure 3.17	A sample help window	93
Figure 4.1	Parent and Child Windows	164
Figure 4.2	Parent/Child relationships of Previous Diagram	165
Figure 4.3	Action Bar	303
Figure 4.4	Reformatted Action Bar	303



---

# Preface

---

The *Microsoft Operating System/2 Windows Presentation Manager Reference*, Volumes 1, 2, and 3, is derived from the latest draft of the functional specification of the Windows Presentation Manager. Although this documentation does not represent the final Windows Presentation Manager specification, it does provide a reasonable preview of the functionality you can expect from the final product.

This documentation is preliminary in nature. The application program interface and other features of the Windows Presentation Manager described in this document are subject to change. It is strongly recommended that the documentation be read for informational purposes only.



---

# Chapter 1

## Introduction

---

1.1	Introduction and Guide to Windows Presentation Manager	3
1.1.1	What is Presentation Manager?	3
1.1.2	Fundamental Features of Windows Presentation Manager	3
1.1.2.1	User Interface Shell	4
1.1.2.2	Screen Appearance.	5
1.1.2.3	The Pointer	7
1.1.2.4	Presentation Manager Windows.	7
1.1.2.5	Presentation Manager User Controls	9
1.1.2.6	Presentation Manager Programming Functions	10



## 1.1 Introduction and Guide to Windows Presentation Manager

This section introduces Windows Presentation Manager to the end user.

### 1.1.1 What is Presentation Manager?

The Windows Presentation Manager is the presentation services component of the MS OS/2 operating system. Its features include:

- the ability to view output from multiple applications on the display simultaneously
- an enhanced User Interface to both MS OS/2 and application functions
- programming interfaces which provide applications with sophisticated functions:
  - for generating and displaying Graphics and Alphanumerics data on a range of output devices including the display screen.
  - for handling Input devices such as mouse and keyboard
  - for Windowing data onto the display screen
  - for the provision of a User Interface which is both rich in function and consistent across applications.

Applications which run with the MS OS/2 kernel will also run when Presentation Manager is present. However, not all these applications can take advantage of the additional facilities provided by Presentation Manager. In particular, applications which attempt to access the display or input devices directly cannot share the screen concurrently with other applications and cannot use the Presentation Manager programming interfaces. The applications which cannot take advantage of Presentation Manager are termed Non-Presentation Manager Applications. The section, "Running MS OS/2 Kernel Applications Under Presentation Manager", defines the applications that may not be run in the Presentation Manager screen group.

### 1.1.2 Fundamental Features of Windows Presentation Manager

### 1.1.2.1 User Interface Shell

When the MS OS/2 system is started up with Windows Presentation Manager present, the display screen is initially occupied by the Presentation Manager User Interface Shell. The Presentation Manager User Interface Shell replaces the simple User Interface Shell provided with the MS OS/2 kernel. It provides the following end-user functions:

---

#### Start an Application

The user is presented with a list of all the available applications and can choose one to start. There is a 'command line' option, which enables the user to start a program by entering the command line in a manner consistent with MS OS/2.

A means is provided for the user to update the list of applications - adding or removing entries as desired - and updating the application profile for each of them.

#### Switch to another Application

The user can display all the applications which are running and can select which one to work with next. The list includes both Presentation Manager and non-Presentation Manager applications.

#### Control of the Position and Size of Application Windows

Each Presentation Manager application has one or more Windows on the screen. The User Interface Shell provides the user means of controlling the size and position of the windows visible on the screen.

#### Control of the Printing functions.

A menu is provided to give the user control over the Printing functions performed by Presentation Manager.

#### Use of MS OS/2 file system

An easy-to-use method of interacting with the MS OS/2 file system is provided, that allows the end-user to perform file commands such as copying or renaming files.

#### Control functions

Provides the user with a consistent and easy-to-use method of selecting defaults for various Presentation Manager parameters, e.g. the colour of empty space on the screen.

### 1.1.2.2 Screen Appearance.

As applications are started by the user they appear on the screen. The applications fall into two classes - Presentation Manager and Non-Presentation Manager. For Presentation Manager applications, the User Interface Shell menus remain visible until explicitly removed by the user. For non-Presentation Manager applications, the User Interface Shell disappears when the application is on the screen.

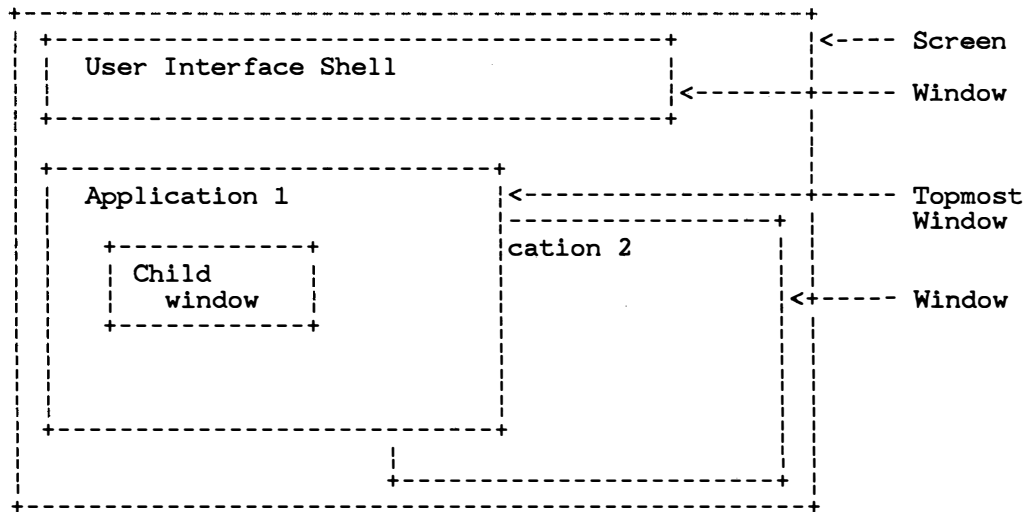
Non-Presentation Manager applications are not able to take advantage of the features of Presentation Manager. The section, "Running MS OS/2 Kernel Applications Under Presentation Manager", defines a non-Presentation Manager application.

Presentation Manager applications are able to take full advantage of the features of the Presentation Manager functions. These applications do not have to use the Presentation Manager unique programming interfaces but do have to obey rules concerning their use of the display screen and the input devices. Put simply, when using the display and input devices an application must use the Presentation Manager programming interfaces and/or use the basic MS OS/2 VIO., KBD., or MOU. function calls.

When the user wants to interact with a non-Presentation Manager application, the application always appears on the screen by itself. Non-Presentation Manager applications cannot share the screen with other applications. Neither can they share the screen with the User Interface Shell. Thus the application cannot be seen at all when the user is interacting with another application or the User Interface Shell.

The User Interface Shell and all the Presentation Manager applications occupy the Presentation Manager Screen Group. They can all potentially appear on the screen simultaneously, in overlapped Windows. A Window is a rectangular region on the screen within which application data is displayed. The Presentation Manager screen has a 'Messy Desk' appearance in that the rectangular windows can overlap one another. Where the windows overlap, only part of one window is displayed and the appearance is like that of papers on a desktop - ie. one piece of paper overlays another and only the topmost one can be seen where they overlap.

A simple example of the Presentation Manager Screen Group appearance is shown in the following diagram.



**Figure 1.1 Typical Presentation Manager Screen Layout**

A Presentation Manager application generally has one window and can have many more. Windows are organised on a hierarchical parent-child basis. A child window always lies on top of and is contained within its parent window. The windows at the top of the structure (which can be thought of as children of the physical screen) are called *top-level* windows. An application may have one or more top-level windows.

The top-level window with which the user is interacting is called the *active* window. This will lie visually on top of all other top-level windows. Keyboard input is always directed to the input focus window. The input focus window is either the active window or a child of the active window.

The mouse input is generally directed to the window that lies underneath the mouse pointer.

Some user input is received by the User Interface Shell rather than an application. This input generally performs operations beyond the scope of a single application, such as allowing the user to switch the active window. Certain keys on the keyboard and the mouse cause this kind of input. A detailed description is provided in the section dealing with the User Interface Shell.



### 1.1.2.3 The Pointer

Part of the screen appearance related to input is the Pointer. The pointer is a small image which moves around the screen as the mouse is moved. It is displayed only on those systems which have a mouse attached. It appears on top of anything else displayed on the screen.

Its appearance can vary. There is a System Pointer appearance, an arrow, which the pointer has by default. The shape can change when the pointer enters an application window. The pointer shape can also vary as it moves from selectable to non-selectable items on the screen.

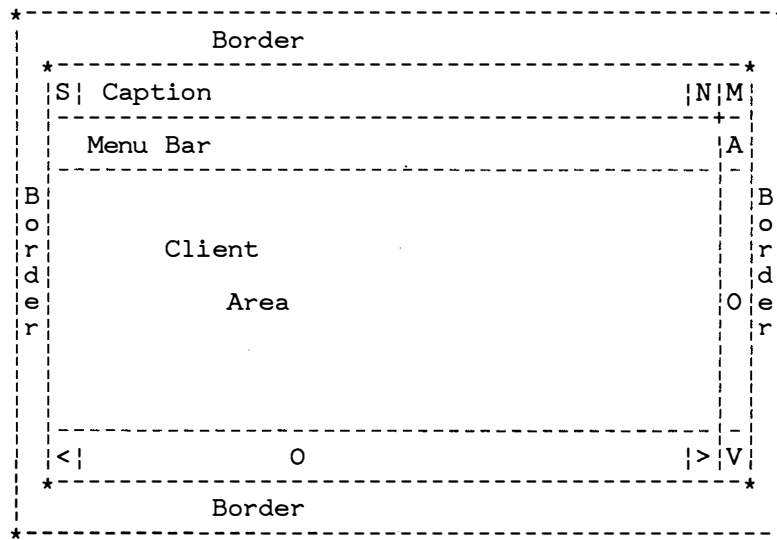
The position of the Pointer on the screen is termed the Action Point. The Pointer can be used to select objects by positioning the pointer over the object and pressing and releasing one of the mouse buttons. Since the pointer is generally a large object, the action point occupies a point within the pointer shape. This point must be chosen carefully to avoid confusing the user. For instance, the action point of the System Pointer is at the tip of the arrow.

### 1.1.2.4 Presentation Manager Windows.

Presentation Manager windows are more than just simple rectangles on the screen. They have a number of optional features which occupy their borders, termed the *frame window*. The frame window gives the end-user access to a number of Presentation Manager functions. The frame window includes:

- Borders
- Caption
- Scroll Bar
- Menu Bar
- System icon
- Maximize and minimize icons

The area in the centre of the window that would normally contain the main information content of the window is called the *Client area*.



S is the system icon  
M is the maximize icon  
N is the minimize icon  
O is the thumb mark in the scroll bars

**Figure 1.2 Presentation Manager window with frame**

### Window Border

Presentation Manager windows have a border in one of four formats:

- Normal border (that is not selectable by the user)
- Heavy border (that is selectable by the user for operations such as sizing a window)
- A thin border (that is not selectable)
- No border

### Caption

The caption is the window name that appears at the top of a window. Highlighting of the caption bar indicates the window with which the user is currently interacting.

### Scroll Bars

A window can contain one or two optional Scroll Bars. There is a Vertical Scroll Bar which appears at the right of the window and a Horizontal Scroll Bar which appears at the

bottom of the window. The scroll bars can be used to move the data appearing in the window up and down or right and left, under either application or Presentation Manager control.

#### Menu Bar

A menu bar is a horizontally aligned menu at the top of the window. The end user may make selections on the menu bar that either send commands directly to the application, or cause the selection of a pull-down menu.

#### System icon

The system icon is an icon that the user may select in order to activate the system menu for the window. The system menu contains functions such as move, size, ...

#### Maximize icon

The maximize icon is an icon that the user may select in order to change a window to its maximum size.

#### Minimize icon

The minimize icon is an icon that the user may select in order to change a window to its minimum size.

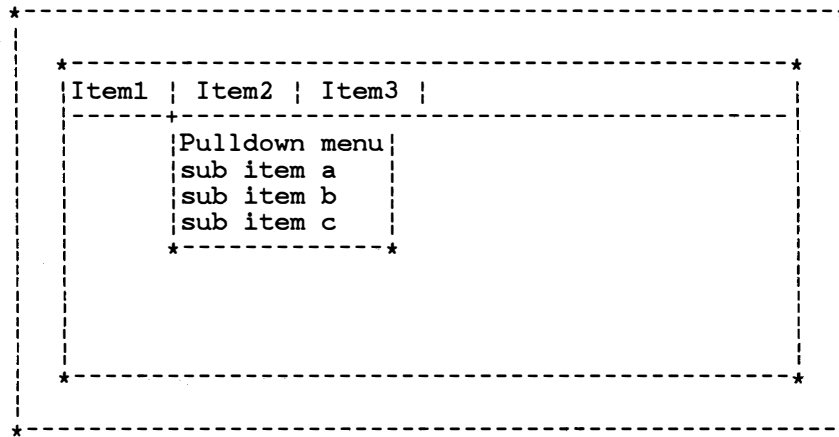
### 1.1.2.5 Presentation Manager User Controls

The Presentation Manager User Controls provide the application program with consistent means of interacting with the user to perform various standard operations. These are:

- Interaction by use of menus
- Interaction by use of dialog boxes

#### 1.1.2.5.1 *Use of menus*

The use of menus to interact with an application will always commence with a menu bar. The menu bar is a horizontal bar along the top of a window that contains a number of items. The items may be selected, one at a time, by the user. The selection of an item in the menu bar by the user will cause the appearance of a secondary menu, called a pull-down menu. The pull-down menu contains additional options, one or more of which may be selected by the user. On completion of the selection, the pull-down menu is removed and the application performs the required action.



**Figure 1.3 Menu bar with pull-down menu**

#### *1.1.2.5.2 Use of dialog boxes*

#### **1.1.2.6 Presentation Manager Programming Functions**

Presentation Manager has a large Application Programming Interface which is subdivided into major functional groups:

- Windowing - creation and control of windows within an application
- Input and Message Handling
- User Controls
- Alphanumeric Output
- Graphics Output
- Bitmaps
- A programmed interface to the User Interface Shell.

It is not necessary for an application to use any of the Presentation Manager API functions in order to run as part of the Presentation Manager Screen Group and be windowed onto the screen with other applications. An application using the VIO., KBD., and MOU. functions of basic MS OS/2 can be windowed when Presentation Manager is present. No changes to the application are necessary.

However, an application using the Presentation Manager API has access to a range of powerful functions which can enhance the functionality and usability of the application while at the same time reducing the effort required to produce it.

A summary of the groups of functions in the Presentation Manager API follows. A thorough description of the functions and their uses is given in later sections.

---

## Windows

An application can create and use a number of windows on the screen via the Windows API. Function is provided to control the size and position of a window and also to control whether the user can size or position a window. The application can specify the form of the window frame. The application can also control the data which appear in each window and can control which window is the Input Window.

## User Interface Controls

The User Interface Controls API provides the application with functions for dialog between application and the user. The functions include:

- The display and interaction with menus.

The following menus are supported:

- Menu Bars
- Pull-down menus
- Control functions that an application would typically group together into a 'dialog box'. These are:
  - Scroll bars
  - Buttons
  - Edit controls
  - Static controls
  - List boxes
  - Message boxes

## Input and Message Handling

The Input API allows the application to control the input it receives, both from the user via the Mouse and Keyboard and from the system and other applications in the form of messages. The input is based on an application input queue, and one or more Window processing functions.

## Alphanumerics Output

The Alphanumerics output API, termed Advanced Vio, is used to output simple Alphanumeric data into screen Windows or into Bitmaps. Advanced Vio is an extension of the basic MS OS/2 VIO.. functions for a windowing

environment. Advanced Vio also allows use of multiple fonts and features such as underscoring of individual characters.

### **Graphics Output**

The Graphics API, called the GPI, is used to draw graphics data into screen windows, bitmaps, or other devices such as printers and plotters. The application can draw a range of graphics objects, such as Lines, Arcs, Text Strings, Closed Areas and Images. Various attributes of the primitives such as their Colour, Area Fill pattern, Character Font and Line Style can be controlled. The size, orientation and position of every primitive can be varied by means of Transformations.

The GPI also supports a wide range of text functions, including the support of fonts.

Graphics data may be managed by the application or stored and managed by the Presentation Manager system.

### **Bitmaps**

The Bitmap API allows creation and use of Bitmaps. Bitmaps are best thought of as images similar in form to the screen image. Bitmaps can be drawn into in a similar fashion to the screen; they may reside either in PC memory or in memory associated with a particular device. Bitmaps can also be the source of data to place on the screen. They can be used to produce rapid changes to the screen, such as changing a Menu, in cases where normal drawing would be too slow.

### **User Interface Shell API**

Presentation Manager contains an API that will allow applications to request some of the shell functions normally requested by the user.

---

# Chapter 2

## Application Model

---

2.1	How to Write a Windows Presentation Manager Application	15
2.1.1	The Purpose of the Presentation Manager API	15
2.1.1.1	Presentation Manager Basic System Structure.	17
2.1.2	API - General Features	20
2.1.2.1	Output Fundamentals	21
2.1.2.2	Input Fundamentals	23
2.1.3	Presentation Space, Device Contexts and windows	24
2.1.3.1	Presentation spaces	24
2.1.3.2	Device Contexts	25
2.1.3.3	Windows	26
2.1.4	Presentation Manager functions	27
2.1.4.1	Output via GPI or Advanced Vio functions	27
2.1.4.2	Output via User Controls functions	28
2.1.4.3	Input Functions	29
2.1.5	Sample Programs	30
2.1.6	Application Model	30
2.1.6.1	Basic Application structure.	31
2.1.6.2	The System Environment - The Shell and Other Applications.	33
2.1.6.3	Program Structure and Windows - Window Pro- cedures.	34
2.1.6.4	Application Rules And Conventions	34
2.1.6.5	Building a Presentation Manager Application	35
2.1.7	Background to user interface	37
2.1.8	Naming Conventions	38

---

2.1.8.1	Constant names	38	
2.1.8.2	Type names	38	
2.1.8.3	Variable and argument names	38	
2.1.8.4	Assembly language structure fields	40	(
2.1.8.5	Standard Data Types Used in this Document	40	
2.1.9	Return Code Conventions	41	
2.1.10	Error Conventions	42	
2.1.10.1	Error Severity	42	
2.1.10.2	Error codes	43	



## 2.1 How to Write a Windows Presentation Manager Application

This section describes how to write a Windows Presentation Manager application. It describes the environment in which a Presentation Manager application runs and gives a guide to the concepts and methods of using the Presentation Manager API. A detailed description of the Presentation Manager API is in the later sections.

### 2.1.1 The Purpose of the Presentation Manager API

The basic purpose of the Windows Presentation Manager is to provide easy accessibility for the user to the functions provided by the PC system. It does this in conjunction with the MS OS/2 kernel and together they help the user accomplish whatever tasks need doing, as and when they are needed.

An important feature of MS OS/2 is that of multi-tasking. The system can perform a number of tasks simultaneously - multiple application programs can run at the same time. Presentation Manager allows the user to see the data belonging to many applications simultaneously, so that one set of data can be used in conjunction with another.

Presentation Manager makes it easy to get things done. In contrast to the situation on MS-DOS, there is no need to stop one application program just because the user needs to run another to get access to some piece of information. Presentation Manager provides means to start any task at any time. It provides means to view many tasks simultaneously on the screen. Presentation Manager also provides means for copying data from one task to another ("Cut and Paste") which really makes the use of multiple tasks interesting and useful.

For application programs, Presentation Manager provides *shared* access to the general resources of the PC system, which include:

- The Screen
- The Keyboard
- The Mouse
- Printer(s)
- Plotter(s)
- Picture Files
- Other Applications

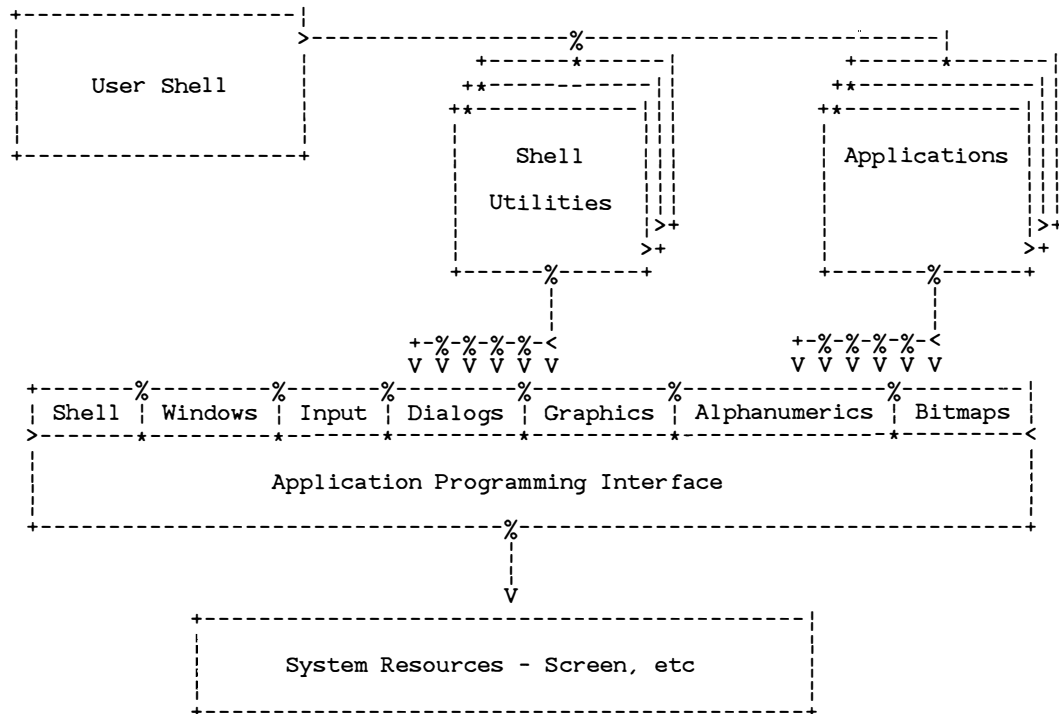
Presentation Manager makes the access to the resources simple.

Presentation Manager also simplifies the process of writing an application through the functions provided in the API and the various utility programs provided in the Toolkit.

The Presentation Manager API does require applications to behave in certain ways, in order to share resources effectively. In simple terms, applications wishing to take advantage of the Presentation Manager features must behave in a *Co-operative* fashion. Applications must co-operate with both the Presentation Manager system and other applications in order that the system works to the benefit of the end-user.

The Presentation Manager API is structured around these basic ideas and does make some demands on the way applications work. This is made clear in later sections.

MS OS/2 applications that do not wish to take advantage of the Presentation Manager facilities can run in a PC system that is using Presentation Manager. How such applications run in a system using Presentation Manager is dealt with in a later section.



- get tasks (applications) running
- work with a particular application
- control the layout of applications on the screen, including position and visibility
- view and work with the data files in the system
- control the Printer(s) and Plotter(s) attached to the system
- control various aspects of the appearance and operation of the system, such as the screen colors

The User Interface Shell is designed to make the system easy to understand and easy to use. It gives rapid access to the capabilities of the system. At the same time, it is functionally very rich and caters for the expert user. Applications should be written with the functions of the User Interface Shell in mind - to avoid unnecessary duplication.

It is important that the various parts of the User Interface Shell and the application programs in the system have the same Style. This means that they are uniform in appearance and all work in the same kind of way, even though the function provided by different applications may be very different. This means that the user does not have to jump from one environment to another and can proceed with ease from one task to the next. In fact, the user should not really be aware of moving from one application to another.

The Presentation Manager API makes it easy for a program to conform to a standard Style. This is discussed in detail in a later section.

Thus, the User Interface Shell provides access to the system, to various utilities and to the applications installed in the system.

#### *2.1.1.1.2 The API*

Applications access the functions of the Presentation Manager system via the API. The API and its associated Utilities simplify the process of writing an application. It provides the following broad areas of function:

- Display of *Data* on the *Screen* and on *Printers* and *Plotters*. The data may be simple Text ('Alphanumerics') or Graphics (including high quality Text).
- Presentation and Operation of Standard User Menus and Dialogs on the screen to aid the user in accomplishment of some task.
- Interaction with other Functions or Applications in the system, including Shell processes and functions.
- User Interaction and Input functions.
- Partitioning of Screen data, economical use of Screen area and structuring of application functions.

The description of the API is divided into a number of functional areas:

- 
- |       |   |
|-------|---|
| Shell | access to aspects of the User Interface and to the Utilities that form part of the User Interface Shell, including: |
|       | <ul style="list-style-type: none"><li>• Starting of Programs - Program Names</li></ul>                              |

- Listing of running applications
- Clipboard - copying of data between programs
- Program environment information - initial values for position and size of an application, for example.

#### Windows

involves provision of areas on the screen in which to draw data. However, the function is much more extensive than this, and touches on:

- program structuring including object-oriented programming
- user interface functions
- user input
- inter-program communication.

#### Input

which covers:

- user input from Mouse and Keyboard
- system messages and inter-application messages
- timer functions

#### Dialogs and Menus

which includes:

- Display and operation of Menus offering the user straightforward selections from a list of items.
- Creation, display and operation of Dialogs which offer the user more complex forms of interaction with the application

#### Alphanumeric Output

which is the output of simple textual information to the screen, printers, plotters and files. This offers a way of displaying text in a simple form as fast as possible.

#### Graphics Output

which allows the application to create and display graphical data on the screen, printers, plotters and files. This includes high quality and high function typographical text functions.

#### Bitmaps

which allow the creation of bitmapped graphical images for the purpose of rapid manipulation of the appearance of the screen.

## 2.1.2 API - General Features

The Presentation Manager API provides functions for the Interface between an application program and the user sitting in front of a PC. For most applications, this means:

- *Output Functions* for the presentation of data of various forms on the Screen in a consistent manner.
- *Input Functions* for the handling of user requests and responses.

Devices other than the screen are also supported for data output, such as Printers, Plotters and Data Files. However, these devices do not participate directly in the interface between the application and the user.

Presentation Manager organizes its user related functions, both output and input, around *Screen Windows*. An application can create as many windows as it desires. Each window serves a part in the dialog between application and user. One window at a time is the center of attention for the user, although other windows may be visible and convey useful and important information.

The way in which an application uses the Presentation Manager API is summarized in.

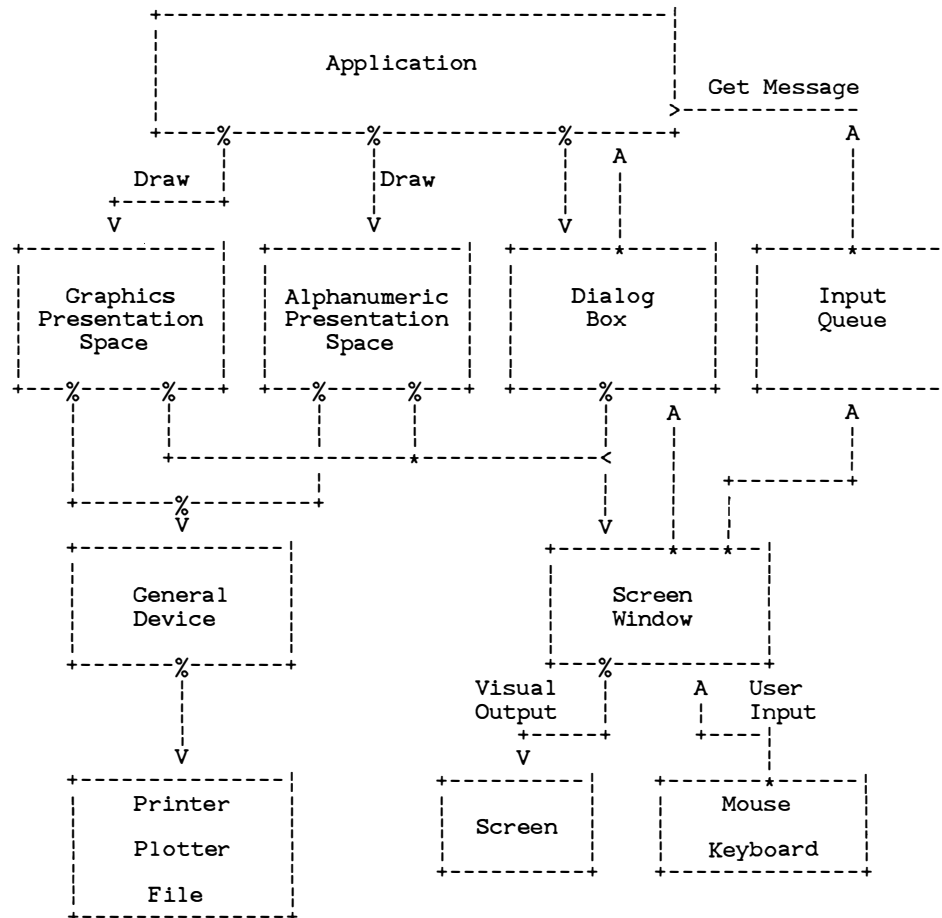


Figure 2.2 Application Model for Input and Output.

### 2.1.2.1 Output Fundamentals

#### 2.1.2.1.1 Output Data

The application creates output data in one of three forms:

##### Alphanumeric Data

which is Text and Numeric data displayed on a fixed grid of character 'slots'. These are held in an Alphanumeric Presentation Space.

Graphics Data

such as lines, circles and shapes filled with colored patterns. These are held in a Graphics Presentation Space.

Dialog Data

which includes items which the user can Select or Type into as part of a structured dialog between application and user. This is held in a Menu or Dialog Box. This data is only displayed on the screen and not on other devices such as printers since it relates directly to the user interface.

Each type of output data has its own set of functions for creating and modifying the data. The data is essentially independent of the place where it is eventually drawn - it is a logical representation of what is required. Thus, for example, a picture can be created in a Graphics Presentation Space, first drawn onto the screen and then drawn onto paper by a printer.

An application may have many instances of each type of data, depending upon the application's requirements. For example, an application would have many Menus and Dialog Boxes if it needed a lot of structured input from the user.

*2.1.2.1.2 Devices*

Output data is drawn onto a *Device*. For Dialog Data, the device is always the Screen. For Graphics and Alphanumerics data the application must *Associate* the Presentation Space with a Device. In these cases, the Device may be the Screen, a memory Bitmap, a Printer, a Plotter or a File. The association can be changed by the application so that the same data can be directed to a number of places in sequence, typically following user requests.

A device is logically represented by a *Device Context*, which encompasses the *Device Driver* required to use the device, and *State Data* which includes appropriate physical realizations of device dependent objects such as Text Fonts.

*2.1.2.1.3 Screen Windows*

Output data is drawn onto the Screen through one or more *Windows*. Each separate Presentation Space or Dialog Box is normally shown in its own window. The windows are created by the application. Windows are rectangular in shape and are fitted onto the screen in a '*Messy Desk*' arrangement. This means that the windows are treated like a series of rectangular pieces of paper on a desk. The windows can overlap one another. Where they overlap, only one of the windows can be seen - the windows have an ordering where one window lies 'on top' of another. Where they overlap, the 'topmost' is seen.



The screen displays all the currently visible windows of all the applications that are running in the Presentation Manager screen group. An application can control the ordering of its own windows relative to one another. It does not control the ordering of its windows relative to the windows of other applications. This is done by the Presentation Manager system according to user requests.

Windows are not used on devices other than the screen. This is because their main use is in enhancing the end-user interface. The screen is special - it is used in a highly interactive way and space for the display of important information is limited. Multiple applications can use other output devices serially, but in a highly interactive environment it is important for the user to be able to see and use multiple applications simultaneously. Similarly, objects such as Menus and Dialogs are used for short periods at a time and should not occupy screen space except when needed. Thus they are placed in windows which can be made invisible.

As well as being a place where an application can display data on the screen, windows have a User Interface aspect as well. The user can alter the position and/or the size of some (but not all) windows. This allows the user to layout work on the screen in a convenient way. To achieve this function, windows have a variety of *Controls* which occupy their borders and allow the user to manipulate the window in a number of ways.

### 2.1.2.2 Input Fundamentals

#### 2.1.2.2.1 User Input

The end user of the system creates input using the Mouse and Keyboard devices. The user can create the following Input Events:

- Mouse Button up/down
- Mouse Movement
- Keyboard Key up/down

Each input event is called a *Message*. User input is asynchronous to applications - that is, the user can press keys or move the mouse to create input independently of the speed with which an application can process the input. All the user's input is buffered as a sequence of Messages in an *Input Queue* before reaching the application. This ensures that input is not lost and is correctly sequenced. The application reads the input messages from the queue one at a time using the *Get Message* or *Peek Message* functions.

There is a close relationship between user input and windows. The user directs input to one window at a time. Each input event is tagged with the ID of the window to which it is directed. Every window is associated with one input queue. Thus input related to a particular window can only be received by reading a particular queue. A single queue can receive input for any number of windows, however.

#### *2.1.2.2.2 Other Kinds of Input*

Input other than Mouse and Keyboard messages can also appear on an input queue. This includes:

- Timer messages, which occur after an application-set Timer expires
- System messages, which inform the application of various system related events. A typical system message is the Paint message, which informs the application that a window (or part of a window) needs to be repainted/redrawn. This often occurs when some or all of the window becomes visible as a result of the user performing a windowing operation.
- Inter-application messages, which are sent from one application to another. These typically occur between applications which are cooperating in some way. Such messages have application-defined meaning.

### **2.1.3 Presentation Space, Device Contexts and windows**

#### **2.1.3.1 Presentation spaces**

A presentation space contains the device-independent quantities required to perform output to an individual window or device. These include:-

- A definition of the picture data itself.  
For VIO output, this is the VIO buffer. For a graphics picture, this could be a graphics segment store (though if non-stored processing is being used, segments are not kept by the system).
- Clipping region as defined by the application.
- Definition of any fonts required for drawing.  
This is essentially a logical description of the fonts, and does not include any physical font definitions.
- Co-ordinate mapping.  
An indication of how world co-ordinates are to be mapped to the device.

- A definition of the colors an application would like.
- The default attributes associated with the picture.

A presentation space is always required whenever the application wishes to use any of the GPI or Advanced Vio functions to output data on an output device or into a bitmap. All of the GPI and Advanced Vio calls require the presentation space handle to be specified as a parameter. The presentation space is created by the VioCreatePS or GpiCreatePS functions.

Before a presentation space can be used to draw a picture, it must be associated with a Device Context. (Refer to following section on Device Contexts.) After this has been done, any drawing operations issued to the presentation space cause output to occur on the device defined by the Device Context.

The presentation space can subsequently be associated with a different Device Context, and the picture redrawn on that device. Because all of the 'application intent' information is kept in the presentation space, the system is able to draw the picture as faithfully as possible on this second device.

Thus a picture which is currently visible on the screen can be printed by temporarily re-associating its presentation space with a Device Context whose device is a printer, and re-drawing the presentation space. In order to continue drawing on the screen, the presentation space is now re-associated back to the screen Device Context.

(Note that the above scenario is only as simple if the entire picture definition has been stored in the presentation space. If non-stored graphics have been used, then the application needs to redraw the picture again after associating with the printer Device Context. However, it still does not need to respecify any of the logical objects, for example fonts, which it needs, since these are still kept in the presentation space.)

### **2.1.3.2 Device Contexts**

A Device Context is the means of drawing to a particular device. It includes a device driver, and also physical realizations, where appropriate, of device-dependent objects which the drawing process requires.

There are four kinds of Device Context, as follows:-

- Screen Device Context. This causes drawing to be performed to a particular window on the screen.
- Memory Device Context. This is used only for drawing to a memory bitmap.

- Metafile Device Context. This causes the picture to be transmitted to a metafile, which may be used to store a picture in editable form.
- Queued device Device Context. This is used for some device other than the screen, for example, an attached printer or plotter, where the output is to go via the spooler.
- Directly attached device Device Context. This is used for some device other than the screen, for example, an attached printer or plotter, where the output is not to go via the spooler.
- Information Device Context. This is used for some device other than the screen, for example, an attached printer or plotter, but where no output will occur. Its purpose is to satisfy queries.

A Device Context is required whenever the application wishes to use any of the GDI or Advanced GDI output functions. However, the Device Context is not normally specified on the GDI or Advanced GDI calls; instead a Device Context is associated with a particular presentation space, by the application issuing a `GdiAssociate` or a `VioAssociate` call (an implicit association is also possible in the GDI case). The Device Context must be specifically created by the application in all cases. The application uses the `DevOpenDc` call to create a Device Context for a printer, bitmap, or metafile. In the case of the display screen, the Device Context for a window is created by a call to `WinCreateWindowDC` after the application creates the window with a `WinCreateWindow` call.

### 2.1.3.3 Windows

A Presentation Manager window is a rectangular area on the screen. A window contains visual data displayed from a Presentation Space, for example a Graphics (GDI) Presentation Space, which is associated with the window. Alternatively, a window could display data from a dialog box.

The screen can have many windows displayed and these may overlap. Where overlap occurs, the windows have a priority ordering. At any point on the screen, the window with the highest priority gets displayed.

Presentation Manager windows are of two types:

---

#### Main Windows

A Main window has the property of being positioned relative to the screen itself. It is not related to any other window. Operations on one main window do not affect other main windows.

An application can have as many Main windows as it wants, within overall implementation limits.

A main window may be considered to be a child window of the entire screen.

### Child Windows

A Child window has the property of being positioned relative to another window, termed its Parent. The Parent window can have multiple Children. Operations on the Parent window affect the Child. For example, moving the Parent moves the Child and hiding the Parent hides the Child.

A Child is constrained to fit within the client area of its Parent. A Child window always has a higher priority than its Parent, i.e. it cannot be hidden simply by virtue of being underneath its parent. A Child window may have Children of its own.

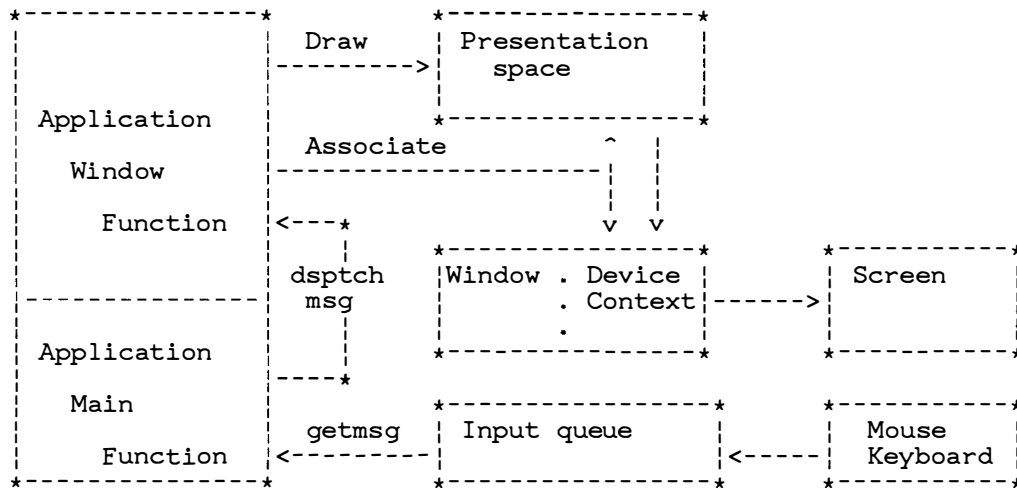
An application can have as many Child windows as it wants, within overall implementation limits.

## 2.1.4 Presentation Manager functions

### 2.1.4.1 Output via GPI or Advanced Vio functions

The data displayed in each window is held in a Presentation Space which is created and manipulated by the application separately from the window. The Presentation Space is different for different types of data - a GPI presentation space for Graphics/Image data, and an Advanced Vio presentation space for alphanumeric data.

The application output does not go directly from the presentation space to the screen window. It goes through a 'Device Context'. The Device Context encapsulates various physical characteristics of the output device. The main utility of the Device Context is for the support of other devices such as printers or memory bitmaps.



**Figure 2.3** Presentation Manager application model for graphics and alphanumerics

Thus to display some data on the screen, the application:

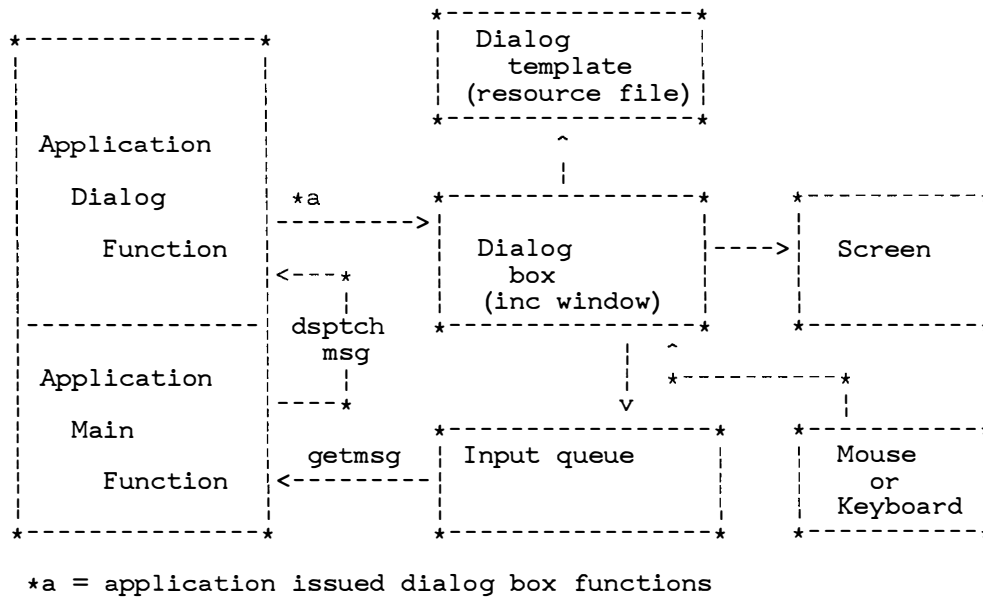
1. creates a Window (this will implicitly create a Device Context that is associated with the window)
2. creates a Presentation Space
3. associates the Presentation Space with the Device Context, so that data drawn from the Presentation Space goes into the Window
4. creates the data to display by operating on the Presentation Space
5. puts the data into the Window by means of a draw operation on the Presentation Space

Note that for Gpi, items 2 and 3 can be combined in a single call, and also in non-stored mode items 4 and 5 are combined.

#### 2.1.4.2 Output via User Controls functions

When an application wishes to use the User Controls functions, it does not specifically create a presentation space. Instead, it interacts directly with a dialog box or menu object. These objects incorporate the concepts of window, presentation space, and Device Context.

In addition, the input that the application receives from the User Controls is processed by Presentation Manager and returned to the application in terms of the particular dialog box or menu.



**Figure 2.4** Presentation Manager application model for dialog boxes

### 2.1.4.3 Input Functions

In Presentation Manager, input to an application is a sequence of Messages generated by any of a number of sources. The Messages are placed on an application Input Queue in time order and are read from the queue by the application. All Presentation Manager applications have at least one input queue - the Default Input Queue. In addition, the application may create additional input queues. Every window created by the application will have a single input queue associated with it. Most input messages are associated with one of the application's windows, and are sent to the appropriate queue.

In addition to an input queue, every window normally has a window processing function associated with it. It is the job of the window processing function to process the input associated with a window. The main program of the application will read the input from the input queue by use of the GetMessage function, and then route the input to the appropriate window processing function by means of the DispatchMessage function.

It is the responsibility of the application to be 'well-behaved'. This means that an application must always issue a GetMessage to read its input queue within a short time period (e.g. 0.1sec) of receiving the previous input. In order to achieve this, it must have dispatched the window processing function, and the window processing function must have completed and returned within the specified time.

If the application does not meet these requirements, various system functions and/or other applications will be locked out until the application completes processing and reissues its read on the input queue.

### 2.1.5 Sample Programs

A number of Sample Programs are supplied with Presentation Manager to help programmers understand the Presentation Manager API and give hints as to how the API can be used to achieve results. Each sample program tackles its own functional area:

- Use of Windows.
- Use of Menus and Dialogs.
- Advanced Vio alphanumerics for display of Text.
- Keyboard and mouse input.
- Graphics:
  - Direct (non-stored) drawing of pictures
  - Stored creation, drawing and editing of pictures
  - Correlation
  - Dragging
  - Bitmap operations
  - Printing
  - Typographic fonts
- Use of the clipboard.

### 2.1.6 Application Model

This section covers various aspects of writing an application to use the Presentation Manager facilities. Presentation Manager places a number of requirements on the way an application is structured and the way in which it uses certain facilities, especially Input and the Screen.

Note that special considerations apply to MS OS/2 applications which use multiprogramming methods, ie. multiple processes or multiple execution threads. These considerations are dealt with in the chapter, "Multiprocessing and Inter-Process Communication"



### 2.1.6.1 Basic Application structure.

To use the Presentation Manager API, a program must call the *WinInitialize* function before any other Presentation Manager function. This function identifies the application to Presentation Manager and initializes the application's environment.

The *WinInitialize* function returns an *Anchor Block Handle* which has the purpose of holding the application's Presentation Manager environment data. The Anchor Block Handle must be stored by the application for later use - it is required by a number of Presentation Manager functions, such as *WinCreateWindow*.

Once *WinInitialize* has been called, the application can use any other Presentation Manager function - to display data on the screen or receive input from the user, for example. If another Presentation Manager function is called before *WinInitialize*, it fails and returns an error code.

When the application is about to finish, it should call the *WinTerminate* function. This is the inverse function to *WinInitialize* - it destroys the application's Presentation Manager environment. After *WinTerminate* has been called, the application cannot make any further Presentation Manager function calls. If the application makes any Presentation Manager function calls after *WinTerminate* has been called, the function fails and an error code is returned.

*WinTerminate* deallocates and destroys any Presentation Manager resources that were allocated to the application, such as Windows and Presentation Spaces. However, it is recommended that such resources are explicitly destroyed by the application before calling *WinTerminate* - this allows the application to perform a tidier 'cleaning up' of the resources, including saving data in disk files if required.

```

Program Fred;
      -|
****  |
****  | >- Program Initialization
****  |
      -+

WinInitialize( );    --- Presentation Manager Initialization

      -|
****  |
****  | >- Main body of the program.
****  |
      -+

WinTerminate( );    --- Presentation Manager Termination

      -|
****  |
****  | >- Program Termination
****  |
      -+

End Program Fred;

```

**Figure 2.5 Application Structure.**

#### *2.1.6.1.1 Normal and Abnormal Application Completion.*

An application which uses the Presentation Manager API normally allocates various resources to itself, such as Windows, Presentation Spaces and Input Queues. It is recommended that the application deallocates and destroys all these resources before it finishes.

If the application fails to deallocate any Presentation Manager resources before finishing, for example by failing to call WinTerminate, then these resources are still deallocated by the system. This occurs, in DOS terms, when the Exit List processing occurs. *Note:* This applies whether the application finishes normally or abnormally (due to some error). The Presentation Manager system ensures that no resources are left 'lying about' once the application finishes.

It is better if the application explicitly destroys any resources since the application can do things in a more coherent order - especially from the appearance of things on the screen. The application can also save away any data associated with the resources, if necessary.

### 2.1.6.2 The System Environment - The Shell and Other Applications.

In Presentation Manager, an application does not stand on its own. It is part of a system which interfaces to the user. In particular, the User Interface Shell forms a major part of the interface and it is through the User Interface Shell that the application is started and is accessed when running.

#### 2.1.6.2.1 Starting an Application

When the user starts an application, this is done via some selection(s) from windows in the User Interface Shell. The application is represented there by a *Long Name*, which is more meaningful to the user than an eight letter filename.

Once an application is running and it creates and displays a main window, the window must be given a Title, so that the user can identify the application.

The User Interface Shell also has a Switch List containing the names of all the main windows of applications in the system. This allows the user to find an application of interest when the system is running more than one application on the screen and some of the applications are obscured by other applications' windows. *It is the application's responsibility to put its entry into the Switch List.*

The Long Name by which an application is started can be found by calling the *WSHGetStartupName* function, which is part of the Shel API. It is recommended that an application uses this name for its main window and for its entry in the Switch List. However, where the application is working with a particular data file, it is also recommended that the file name is appended to the Long Name to form the Window Title and the Switch List entry.

#### 2.1.6.2.2 Main Window Title

The title of the application's main window is set in the *WinCreateFrameWindow* function when the window is created. However, the title can be updated subsequently by sending a *WM\_SETWINDOWPARAMS* message to the window, for example if the application starts work on another file.

#### *2.1.6.2.3 Switch List Entry Name*

The Switch List entry is created by the *WSHAddSwitchEntry* function. The Name displayed in the Switch List is specified as a parameter to this function, along with the Window Handle of the application's main window. When the user selects the Switch List entry belonging to the application, the main window is made Active and it is brought to the top of the stack of windows.

#### **2.1.6.3 Program Structure and Windows - Window Procedures.**

A Screen Window is not only used as a place to put display data. Windows have an important role in a number of aspects of the Presentation Manager API:

- Display of multiple sets of data on the screen.
- Efficient use of scarce screen area.
- Handling of Input - both from the user and the system.
- Program structuring and partitioning.
- A seamless way of extending System functions.

#### **2.1.6.4 Application Rules And Conventions**

##### *2.1.6.4.1 Mouse Button Activation of a Window*

It is the application's job to transfer active status to one of its windows if it gets a mouse down message. It should do this by calling *WinSetFocus()* or *WinSetActiveWindow()* with the window that the mouse message was sent to.

##### *2.1.6.4.2 Active Windows, Dialog Boxes and User Expectations*

The application should not call *WinSetActiveWindow()* arbitrarily to set the active status to one of its windows. This should only be done as the result of an explicit user action requesting a new window to become the active one or as the result of a message from the shell to the same effect.

Neither should an application display a dialog box or message box arbitrarily if it needs to tell the user something and it doesn't own the active window. Applications that need to do this should call *MessageBeep()* a few times and then call *FlashWindow()*. *FlashWindow()* will start the frame of the window flashing. The user will hear a beep and see the window flashing. The user can then choose to pay some attention to the application, and request that it become the active one, say by clicking the mouse with

the pointer in the flashing window. The application can then call `FlashWindow()` again to turn off the flashing, and bring up an appropriate Message or Dialog.

#### *2.1.6.4.3 Mouse Cursor Shape within a Window*

It is the application's job to set the shape of the mouse cursor when it gets a `WM_MOUSEMOVE` message, using the `SetCursor()` call. Child windows should send the `WM_CONTROLCursors` message to their parents, so their parents can have the choice of setting their cursor shapes. See 'Control Manager'.

### **2.1.6.5 Building a Presentation Manager Application**

This section describes the method of building a Presentation Manager application. This includes details specific to Presentation Manager applications that do not apply to other DOS applications. The reader is assumed proficient in building general DOS applications.

The following describes the source files required for Presentation Manager, and the processes through which these are turned into an executable file. The application programmer is responsible for providing three types of source files:

- a resource file
- one or more source code files (i.e. C or assembler files)
- a DOS module definition file.

#### *2.1.6.5.1 Resource Files*

The resource file contains descriptions of the application's user interface data, such as dialog boxes or menus. The application programmer defines these either through a text description, or by using a tool such as the dialog editor which will in turn create the text description.

The Resource Compiler understands these descriptions, and performs two functions in building an application. First, it compiles the text description into a binary format suitable for the Presentation Manager system. Second, it inserts these binary resources into the executable file. The insertion must be done after linking the objects, i.e. the sequence is:

```
link  
rc
```

The resource compiler is invoked through the command:

```
rc resourcefilename [exefilename]
```

The resourcefilename is the DOS filename of the resource text file. If no extension is specified, the extension is assumed to be RC. The exefilename is optional, and is the name of executable to insert the binary resources into. If it is not specified, then the default is the executable with the same filename as resourcefilename, i.e.

```
rc sample
rc sample.rc
rc sample.rc sample.exe
```

would all compile the resources described in sample.rc and would insert them in sample.exe.

Compilation of the resources takes time, and the resources must be reinserted in the executable every time the application is relinked. Thus, to save application build time, the Resource Compiler has an option to compile the resources and then create an intermediate object file. This resource object file can then in turn be specified as input to the Resource Compiler, to complete the final step of insertion into the executable.

To create the intermediate object file, specify the "-r" option, which will create a file whose extension is RES.

Example:

```
rc -r sample.rc
link
rc sample.res sample.exe
```

#### *2.1.6.5.2 Source code*

High level language files are compiled using the appropriate language compiler; assembler files must be assembled. In both cases, intermediate object files (.OBJ) should be created.

#### *2.1.6.5.3 Module Definition File*

All external entry points in a Presentation Manager application must be EXPORTed in the Module Definition File (.DEF). See "Building an OS/2 Application" for a further description of the .DEF file.

#### *2.1.6.5.4 Linking A Presentation Manager Application*

At link time, the developer must specify:

- the code object files to be linked (.OBJ's)
- the Module Definition File (.DEF)
- Libraries (.LIB's)

In order to resolve references to Presentation Manager API, the developer must specify Wincalls.lib in addition to any other necessary libraries.

Sample Build Sequence

- `rc -r sample => creates sample.res`
- `Compile sample.c = creates sample.obj`
- `Link sample.obj, sample.def, wincalls.lib => creates sample.exe`
- `rc sample.res => modifies sample.exe`

Sample.exe is now ready to run under Presentation Manager.

### **2.1.7 Background to user interface**

The MS OS/2 user interface is a set of rules intended to provide end users with a consistent, easy-to-use interface across applications.

It includes many elements of user interaction with the system, such as menu selection and text string input, but it does not include interactions specific to applications, such as spreadsheet editing.

Where an application has user interaction in areas covered by user interface rules, it must conform to the user interface.

The principal topics in the user interface are as follows:

1. Key assignments
2. Menu colors
3. Application action bars
4. Pop-down menus
5. Scroll bars
6. Types of selection fields
7. Entry fields

8. Message and Help panels
9. Window sizing and moving.

Presentation Manager allows all applications to conform to the user interface. For some rules, Presentation Manager enforces conformance by taking over complete parts of the operator interaction. Thus, for these interactions, the only way the application could avoid being in conformance would be to rewrite part of the code provided with Presentation Manager.

## 2.1.8 Naming Conventions

Here is a short description of the variable and argument naming conventions used in the Presentation Manager spec. A name is made up of a tag prefix and an optional identifier. The tag is all lower case, and the identifier begins with an upper case letter. You can either make up your own tags for new data types, or use some combination of the standard tags.

### 2.1.8.1 Constant names

All constants are written in upper case. If applicable, constant names have a prefix derived from the name of a function, message, or idea associated with the constant. For example:

WM_CREATE	- Window message (WM_*)
WS_CLIPSIBLINGS	- Window style (WS_*)
DT_CENTER	- DrawText() code (DT_*)

### 2.1.8.2 Type names

Type names are written in upper case. Type names are usually longer and more descriptive than their variable and argument prefixes; for example:

Type	Prefix
RECT	rc
POINT	pt

### 2.1.8.3 Variable and argument names

A name is made up of a tag prefix and an optional identifier. The tag is all lower case, and the identifier begins with an upper case letter. You can either make up your own tags for new data types, or use some combination of the standard tags.



## Standard name prefixes:

p	- near pointer
lp	- far pointer
d	- delta
c	- count
i	- index
rg	- array
f	- boolean
h	- handle
ch	- character
b	- byte
w	- word
l	- long
id	- ID
it	- item
cmd	- command
pfn	- near function address
lpfn	- far function address
psz	- near ptr to zero terminated string
lpsz	- far ptr to zero terminated string
rgf	- 16-bit packed array of flags/bits
lrgf	- 32-bit packed array of flags/bits
brgf	- 8-bit packed array of flags/bits

## Standard type abbreviations:

hab	- Anchor block handle
hwnd	- window handle
rc	- rectangle
pt	- point
hmenu	- menu handle
t	- 32-bit millisecond value
x	- x coordinate
y	- y coordinate
hps	- PS handle
hvps	- VIO PS handle
hdc	- device context handle
hbm	- bitmap handle
hrgn	- region handle
hcsr	- cursor handle
hdc	- DC handle
msg	- window message ID
style	- 32-bit window style

## Standard type identifiers

Next	- Next
Prev	- Previous
First	- First value (used with Last)
Last	- Last value (== last value, not one greater)
Min	- Minimum value (used with Max)
Max	- Maximum value (one past last possible)

Examples:

pch	- Near pointer to a character (or characters)
rgbBuffer	- Array of bytes
dx	- Delta-x value
cyMax	- Max count of y coordinates
rgfMenu	- Menu command values
lpphwnd	- Far pointer to a near pointer to a window handle.

#### 2.1.8.4 Assembly language structure fields

In assembly language, all structure field names must be unique. Since not all structure fields have unique names, the assembly language convention is that all field names are prepended with the structure type abbreviation and an underscore. Here are some examples:

RECT xLeft field:	rc_xLeft
POINT y field:	pt_y

#### 2.1.8.5 Standard Data Types Used in this Document

Below is a list of the standard data types used in this document:

---

Type	Description
CHAR	Signed 8-bit value or character
INT	Signed 16-bit value
LONG	Signed 32-bit value'
UCHAR	Unsigned 8-bit value
UINT	Unsigned 16-bit value
ULONG	Unsigned 32-bit value
LPSTR	Far pointer to a character string
BOOL	16-bit Boolean (zero ==> FALSE, non-zero ==> TRUE)
HANDLE	32-bit handle

SHANDLE  
16-bit handle

FARPROC  
Far pointer to a procedure

Here are the standard declarations in C for these types:

```
typedef unsigned char UCHAR;  
  
typedef unsigned int UINT;  
  
typedef unsigned long ULONG;  
  
typedef char FAR *LPSTR;  
  
typedef int BOOL;  
  
typedef long LONG;  
  
typedef char FAR *HANDLE;  
  
typedef unsigned int SHANDLE;  
  
typedef int (FAR * FARPROC) ();
```

## 2.1.9 Return Code Conventions

This section documents the strategy for return codes used for Presentation Manager.

Each function is allocated to one of the following types, depending upon the ranges of the values which it returns:-

- Restricted range return values

Where these are successful, a value within a specified range is returned. If there is an error, a value outside the valid range (or a special value within the range) is returned. The error code may be obtained by `WinGetLastError`.

Functions which return handles are a form of this type (0 and -1 are assumed to be outside the range of valid handles).

Examples: `WinCreateWindow`, `GpiQueryMix`

- Defaulted return values

Functions for which a documented, reasonable default behavior exists if an error occurs. Errors in these situations are not interesting, and not harmful to applications.

Example: `WinIsWindowVisible()`

- Failsafe unrestricted range return values

These functions tend to be speed critical, and the return value will be returned in AX.

Example: `WinGetCurrentTime`

- No return values, or structure return values, or unrestricted range return values:

Error code (Boolean) is returned in AX; the return value(s) (if any) are returned through parameter(s).

Example: `GpiQueryAttrs`

## 2.1.10 Error Conventions

### 2.1.10.1 Error Severity

Errors fall into one of the following categories:-

---

#### Warning

The function detected a problem but took some remedial action which enabled the function to complete successfully.

#### Error

The function detected a problem for which it could not take any sensible remedial action. The system will be able to recover from the problem, in the sense that the state of the system, with respect to the application remains the same as at the time when the function was requested i.e. the system has not partially executed the function.

#### Severe Error

The function detected a problem from which the system cannot reestablish its state, with respect to the application, at the time when that function was requested i.e. the system has partially executed the function, and therefore necessitates the application performing some corrective activity in order to restore the system to some known state.

#### Unrecoverable Error

The function detected some problem from which the system cannot reestablish its state, with respect to the application, at the time when that call was issued and it is possible that the application cannot perform some corrective action in order to restore the system to some known state e.g. the application provides the address of the anchor block which the system discovers is apparently corrupted.

Severity levels are 16 bit unsigned integers, with the following values:-

SEVERITY_NOERROR	0x0000
SEVERITY_WARNING	0x0004
SEVERITY_ERROR	0x0008
SEVERITY_SEVERE	0x000C
SEVERITY_UNRECOVERABLE	0x0010

### 2.1.10.2 Error codes

WinGetLastError returns a 32 bit value. The format of this value is:

High uint:	16 bit severity level
Low uint:	16 bit error code

The following is a list of errors returned by the window functions. Gpi errors are listed for each call.

The actual codes will be defined later.

```
WINERR_INVALID_SELECTOR
WINERR_INVALID_STRING_PARM
WINERR_INVALID_HEAP_HANDLE
WINERR_INVALID_HEAP_POINTER
WINERR_INVALID_HEAP_SIZE_PARM
WINERR_INVALID_HEAP_SIZE
WINERR_INVALID_HEAP_SIZE_WORD
WINERR_HEAP_OUT_OF_MEMORY
WINERR_HEAP_MAX_SIZE_REACHED
WINERR_INVALID_ATOM_TABLE_HANDLE
WINERR_INVALID_ATOM
WINERR_INVALID_ATOM_NAME
WINERR_INVALID_INTEGER_ATOM
WINERR_ATOM_NAME_NOT_FOUND
WINERR_INVALID_WINDOW_HANDLE
WINERR_INVALID_MESSAGE_QUEUE_HANDLE
WINERR_INVALID_PARAMETER
WINERR_WINDOW_LOCK_UNDERFLOW
WINERR_WINDOW_LOCK_OVERFLOW
WINERR_WINDOW_LOCKED
WINERR_WINDOW_UNLOCKED
WINERR_NO_MESSAGE_QUEUE
```



# Chapter 3

## User Interface

---

3.1	User Interface Shell	49
3.1.1	General Features of the Shell	50
3.1.2	The Pointer	51
3.1.3	Selection Cursor	51
3.1.3.1	Selecting Items	52
3.1.3.1.1	Single Selection	52
3.1.3.2	Multiple Selection	53
3.1.3.3	Extended Selection.	53
3.1.4	Use of Keyboard and Mouse	54
3.1.4.1	Keyboard	54
3.1.4.2	Mouse	56
3.1.5	Functions for Controlling windows	57
3.1.5.1	Appearance of Windows	57
3.1.5.2	The Shell, Windows and Tasks.	57
3.1.5.3	The Input Focus	58
3.1.5.4	Window manipulation - the System Menu.	58
3.1.5.4.1	Z-ordering	59
3.1.5.4.2	Window Maximize	59
3.1.5.4.3	Window minimize	59
3.1.5.4.4	The Parking-Lot	60
3.1.5.4.5	Change Window Size	60
3.1.5.4.6	Window Move	61
3.1.5.4.7	Restore	61
3.1.6	File Cabinet - functions for using Directories and Files	61
3.1.6.1	The File Cabinet Window	63
3.1.6.2	Tree Window	63

3.1.7	File Cabinet functions	66
3.1.7.1	The File menu	66
3.1.7.2	Direct manipulation	69
3.1.7.2.1	Summary of Mouse use in Direct Manipulation.	69
3.1.7.3	Options menu	71
3.1.7.4	Special Menu	73
3.1.7.5	The Window menu	74
3.1.7.6	STARTUP window	75
3.1.7.6.1	STARTUP Functions	76
3.1.8	STARTUP Editor	77
3.1.8.1	The Exit menu	80
3.1.9	Task Manager	81
3.1.9.1	How to Access The Task Manager.	81
3.1.9.2	Jump Ordering	82
3.1.9.3	How to Work With a Task.	83
3.1.9.4	How to Close a Task	84
3.1.9.5	How to Terminate a Task.	84
3.1.10	Control Panel	86
3.1.10.1	Main panel	86
3.1.10.2	Preferences pull down	87
3.1.10.3	Settings pull down	88
3.1.10.4	Configuration pull down	88
3.1.11	Clipboard Viewer	89
3.1.11.1	Clipboard mechanics.	90
3.1.11.2	Copy and paste for VIO applications	90
3.1.12	Initialization	91
3.1.12.1	The initial view of the system	91
3.1.12.2	The Initialization File	91
3.1.13	HELP facility for the shell.	91
3.1.13.1	INVOKING HELP	92
3.1.13.2	THE HELP WINDOW	92



3.1.13.3	HELP INTERACTIONS	93
3.1.13.3.1	F1=General Help	94
3.1.13.3.2	F5=Index	94
3.1.13.3.3	Shell Help Index	94
3.1.13.3.4	F9=Keys	95
3.1.13.3.5	Esc=Cancel	95
3.1.13.4	Additional notes about Help	95
3.1.13.5	Help on items in STARTUP	95
3.2	The User Interface Shell API	96
3.2.1	Definitions of terms used in the Shell API	96
3.2.2	List of functions provided by the Shell API	102
3.2.2.1	Program Use	102
3.2.2.2	Adding a Program.	102
3.2.2.3	Switching Programs.	103
3.2.2.4	Clipboard	103
3.2.2.5	Control Panel	103
3.2.2.6	PrtPlot	104
3.2.3	Using the API for application programmers	104
3.2.3.1	Executing Programs	104
3.2.3.1.1	Starting Presentation Manager programs	104
3.2.3.1.2	Starting non-Presentation Manager programs	105
3.2.3.2	Creating or changing the switch list entry	106
3.2.3.3	Installation of Presentation Manager	107
3.2.3.3.1	Building the input for WSHAddProgram	107
3.2.3.4	Clipboard functions	108
3.2.3.5	Switching to another application	108
3.2.3.6	Using the API File Selection function	108
3.2.3.7	File System functions.	108
3.2.4	Detailed Description of Shell API Functions	108
3.2.5	Program Use API	109
3.2.6	Adding a Program API	112

3.2.7	Switching Programs API	125
3.2.8	Presentation Manager Initialization File and Control Panel AP	137
3.2.8.1	Overview of control panel	137
3.2.9	Presentation Manager Initialization File Functions	138
3.3	Shell API Structure definitions	145
3.3.1	Shell API data structure reference	145

## 3.1 User Interface Shell

The following sections describe in detail the appearance and function of the Presentation Manager User Interface Shell.

In general, the Presentation Manager Shell aims to present on the screen all the functions available in the system. In complete contrast to the simple MS-DOS user interface, where very few of the objects and functions of the system are visible on the screen, the Presentation Manager Shell can show a visual representation of all objects in the MS OS/2 system and the functions which operate on them. The approach used is an 'Object-Action' one, where the user selects an object to work with and then chooses an action to perform on it. Direct manipulation techniques are used by the Shell, such as dragging objects around, and selecting actions from pop-up menus.

One aim of the Presentation Manager Shell is to reduce the need for the user to read manuals. In part this is achieved by the user's ability to see all system function on the screen. Another contribution to this aim is the consistency of use of the input devices - Keyboard and/or Mouse. The user needs to understand only a very few concepts about these devices to use the system. An important part of the Shell is the online Help facility which it provides - again reducing the need for reference manuals.

A further important aspect of the Shell is that its user interface exemplifies the user interface which should be used by applications. In conjunction with the Presentation Manager programming interfaces (which the Shell uses), the Shell encourages the consistent end-user interface which is a prime aim of the Presentation Manager product.

Consistent with these objectives, the Shell sets out to provide the following capabilities:

1. Provide structured access to the files found on the user's system.
2. Provide a simple, intuitive approach to filing system management
  - high and low level topology/index to the filing system
  - basic manipulation operations (copy, rename, etc.)
  - access to files stored in the system with direct manipulation from the mouse
  - visibility of filing system while running applications
3. Allow for the creation of easy-to-use systems - that is, to provide a way to configure a system to allow the naive users to focus and work with a predetermined set of applications and files.

### 3.1.1 General Features of the Shell

The basic functions included in the Shell are:

1. *The Screen Layout* - Windows
2. *The File Cabinet* is the index, viewer, and repository of objects related to the user's data and program storage. Contained within the File Cabinet there are:
  - Drives
  - Directories
  - Programs
  - OS/2 files
  - STARTUP (programs)

The file system is hierarchical - it consists of drives, which contain a mixture of OS/2 files and directories. Directories in turn may also contain such a mixture, yielding a tree of arbitrary depth. By opening a drive or directory in the tree, the user reveals a window showing the objects found in that drive or directory.

*STARTUP*, contained within the File Cabinet, and allows the user to easily manipulate installed applications; for example, Starting a program running.

3. *The Task Manager* is the window that provides access to, and control of, objects that exist in the working environment. In addition, it provides general control of the user's session, defined as that period of time that the user is interacting with the system.

A window in the workspace can easily be brought to the front using the Task Manager, by selecting its name from the list presented in the Task Manager window. In addition, it may also be possible to perform certain other operations on the selected task, such as closing it.

4. *The Control Panel* allows users to set their workstation configuration and other system related parameters.
5. *Printer Services* provides output control for any device other than the screen.
6. *The Clipboard Viewer* is used to view the contents of data in the Presentation Manager clipboard.
7. *Startup Editor* allows new programs to be defined, and existing programs to be changed.

These functions are described in more detail in the following sections.

The initial implementation of the Shell includes direct manipulation in the File Cabinet using the mouse.

The data displayed on the screen is divided into a number of windows. Each window encloses the data belonging to one part of the interaction between the system and the user. The windows are rectangular and can overlap, giving the appearance of papers on a desk top. This means that one window obscures the part of another “underlying” window where they overlap.

The screen can have many windows displayed at once. A particular use made of windows is the display of menus and dialogs, where the window is displayed for a short time while the user makes a choice or inputs some data. The window is then removed to avoid cluttering the screen.

### **3.1.2 The Pointer**

The Pointer is normally displayed only if there is a mouse attached to the system. However, it is also displayed on mouse-less systems at certain times to indicate to the user that some particular action is taking place. For example, an hour glass is shown to indicate that the user must wait while a lengthy operation is in process.

The pointer is a small shape which reflects mouse movements on the screen. The pointer is displayed 'on top' of the other data on the screen and so is always visible. The position of the pointer marks the user's center of interest and activity on the screen. Its position is used when the mouse buttons are pressed; for example, to select an item of data on the screen.

The shape of the pointer can vary as it travels over the screen. There is a system default shape - an arrow - but each window on the screen can have its own pointer shape defined. When the pointer position moves into a window which has its own pointer shape, the pointer changes to that shape. Similarly, when the pointer position leaves the window, the pointer shape changes back to the system shape or to the shape defined for another window.

### **3.1.3 Selection Cursor**

The Selection Cursor is used to indicate a selectable item that the user can select. It is displayed whether or not a mouse is attached and is in addition to the pointer. It marks the whole of a selectable item as being the center of interest for the user. For example, it can indicate one item in a list of menu items which the user wants to select. The Selection Cursor can be moved by the mouse or by keystrokes from the keyboard.

### 3.1.3.1 Selecting Items

Many items on the screen are *Selectable*. This means that the user can choose the item ( *Select* it) and then perform an action on this item.

Selection is performed in a standard way and is not dependent on the kind of item involved. The way of selecting an item differs between the mouse and the keyboard. Generally, a single item is selected at a time and then some function performed. However, in some cases, multiple items can be selected and the same action performed on all of the items. Multiple selection involves a different way of using both the mouse and the keyboard. All these methods are described in the next sections.

#### 3.1.3.1.1 Single Selection

---

##### Keyboard

The user can move the selection cursor around the selectable items in the window which has the input focus. This is done using the arrow keys, which move the Selection Cursor to the next selectable item in the direction indicated by the arrow. To move from one group of controls to another, the Tab key is used.

Movement of the Selection Cursor normally involves auto-selection of the items. That is, when the Selection Cursor is moved, the item onto which the cursor moves is selected and the previous item is deselected.

##### Mouse

The user can move the mouse until the Pointer lies over a selectable item and then press button 1 down. The item is displayed in reverse video. When button 1 is released, the item is selected. Any other item(s) selected are deselected.

The press and release of the mouse button in this way is termed a "click"

The following action occurs for menu bars and pop-downs. The user holds down button 1 and moves the pointer around the screen. When the Pointer moves away from the original item, it stops being shown in reverse video. If the Pointer tracks across other selectable items, whichever item is under the Pointer highlights. This allows the user to browse around the selectable objects. Whichever item is under the Pointer when button 1 is released is selected. If no item is under the Pointer, no selection occurs.

This use of the mouse is termed "press and hold".

For other than menu bars and pop-downs, the mouse button must be pressed to change the selected item.

### 3.1.3.2 Multiple Selection

In fields with multiple selection (e.g. Check boxes), the following applies:

---

#### Keyboard

The user can move around the selectable items in the window as for single selection. The space bar is used to toggle selection of an item, and the Enter key to submit the panel.

#### Mouse

The user can move the mouse until the Pointer lies over a selectable item, and then click button 1. The item is displayed as selected, other selections are unchanged. To deselect an item, the user clicks on it again.

### 3.1.3.3 Extended Selection.

Note that not all windows allow extended selection. Some windows restrict the user to a single selection at a time. This is typical of pull-down menus, for example. For those windows that do allow extended selection, the following methods apply.

---

#### Keyboard

To extend the selection of items using the keyboard, the user must press the space bar to change from the auto-select mode to multiple selection mode. The user can select additional items, by pressing the space bar again.

The mode terminates when Enter is pressed, or the dialog cancelled. Whilst the mode is active, the selection cursor is displayed independently of selected emphasis, even when they apply to the same item.

To select contiguous items, shift+arrow keys may also be used. These do not cause a mode to be entered.

#### Mouse

The user can press the space bar to switch to multiple selection mode. and then click mouse button 1 with the pointer over an item to select it. Shift+button 1 may also be used; this extends selection from the selection cursor to the position clicked on.

Double clicking the mouse performs the default action on all selected items if in extended selection mode. Otherwise, to invoke the default action on all selected items the user must hold the shift key down while double clicking the mouse.

### 3.1.4 Use of Keyboard and Mouse

Keyboard and mouse can be mixed for selecting groups of objects. The only exception to this is that direct manipulation of files is not available from the keyboard.

This section describes the standardized actions and their meanings. It gives the user a guide to using the Presentation Manager system and the programs which run with it. This is done in terms of the keyboard keys and mouse actions which can be used and their meanings in terms of system functions.

#### 3.1.4.1 Keyboard

The keyboard keys are divided into several logical sections:

---

##### Alphanumeric Keys

which include the A-Z, 1-0 and special character keys. These are typically used only for the input of data. Corresponding characters appear on the screen when these keys are pressed and they generally have no other effects.

##### Function Keys

which typically include F1-F12. These are normally used to invoke particular actions. For example, F1 has the standard meaning of 'Help' and brings help information onto the screen.

##### Movement keys

include the Arrow keys (Up, Down, Left and Right cursor movement keys). The Home, PgUp, End, and PgDn keys also fall into this class. These are used to cause objects to move around the screen. The typical object that they move is the Selection Cursor when the user is interacting with a list of selectable items.

##### Ancillary Keys

including the Shift, Alt, Ctrl Keys. These are used to modify the meanings of other keys. The simplest example of their use is to cause the alphabetic keys to produce uppercase characters when the Shift key is held down.

Specific meanings of some of the keys are described below. This list includes all those keys with associated functions which are essential to the use of Presentation Manager:

---



Alt+Esc	Jump to next task/program (includes non-Presentation Manager programs). This makes the current active application inactive and causes the next task in the Task Manager to become active. (Also see the section, "Jump Ordering")
Ctrl+Esc	Jump to Task Manager. Causes the Task Manager to become the active window. The Task Manager list window is brought to the top. This occurs even if the active application is not in the Presentation Manager screen group, in which case the screen group is switched back to the Presentation Manager screen group first.
Enter	This has two meanings depending on context: <ul style="list-style-type: none"><li>• Submit the changes.</li><li>• Take the default action on the selected item(s)</li></ul>
Arrow Keys	Move Selection Cursor to next selectable item. (selects items as it moves, with deselection of previous item(s) - for Auto-Select)
Shift+Arrow Keys	Extended selection - input field. (Swipe and type)
Delete	Deletes selected text to clipboard, or deletes single character to right of insertion point (cursor).
Backspace	Deletes character to left of cursor
Ctrl+Arrow Keys	Moves to the beginning/end of fields, or words.
Spacebar	Toggles selection status of item for multiple selection panels and also switches into extended selection mode.
tab	Moves selection cursor between groups of controls
F10	Toggle to/from (Application) menu bar (same as Alt make/break)
Alt make/break	Toggle to/from (Application) menu bar (same as F10)
Shift+Esc	Bring Up System Menu (or remove, if already shown)
Alt+F4	Close window (if Close is on the System Menu)
Alt+F5	Restore window

- Alt+F7    Move window
- Alt+F8    Size window
- Alt+F9    Minimize window (toggle)
- Alt+F10   Maximize window (toggle)

#### 3.1.4.2 Mouse

The mouse is used in two ways. It can be moved. It has buttons that can be pressed. These actions are used in conjunction to provide a powerful tool with which the user can interact with the system, using the screen to provide rapid feedback.

Some actions cannot be done with the mouse alone. The shift key on the keyboard is used to perform certain actions. The list below shows when the keyboard shift key is used.

- 
- Button 1 Click
    - Select item under Pointer
  - Shift+Button 1 Click
    - Extend selection to include items between pointer and previous cursor position.
  - Button 1 Double Click
    - Select item and perform default action. If in extended selection mode, add item under pointer to selected items, and perform default action on all items.
  - Shift+Button 1 Double Click
    - Add items to Selection between pointer and previous cursor position, and perform default action on all selected items.
  - Button 3 Click
    - Jump to Task Manager
  - Button 3 Double Click
    - Jump to next task (Also see the section, "Jump Ordering")
  - Mouse Movement
    - Moves the Pointer around the screen. It is used to indicate the point of activity or interest in the data presented on the screen.
  - Button 2
    - Application defined meaning - can vary from program to program.

Press and hold button 1

Drag selected object around window or screen. Meaning is context and application dependent.

### **3.1.5 Functions for Controlling windows**

#### **3.1.5.1 Appearance of Windows**

All main task windows are surrounded by emphasised borders when shown at any size except maximized or minimized. Maximized windows may be shown with all borders just off the screen, although conceptually still there, but in this case they may not be moved.

Minimized windows are distinct in that they are shown in iconic form. Hence minimized windows do not have normal borders.

All windows must have a title bar except:

1. Minimized windows
2. Maximized windows which need the whole screen

Minimized windows may be restored by double clicking on the icon.

The title text of the window is shown next to the icon when it has the input focus.

A single mouse click on a minimized window shows the Systemu. A double mouse click (or pressing Enter when the minimized window has been selected) will open the icon up to show the program.

Non-Presentation Manager programs show up as icons in the Presentation Manager screen group. A System Menu is shown when they are selected, but a screen group switch only takes place when the window is opened for use. Thus the user can browse all tasks in the system without constant switching of screen groups. Note that if a non-Presentation Manager program is selected in the Task Manager window, then the program is shown directly, rather than bringing the icon representing that program to the front of the screen.

#### **3.1.5.2 The Shell, Windows and Tasks.**

User input, such as a keystroke, mouse movement or a mouse button press is either passed directly to a program, or is intercepted by the Shell. Input to the Shell may in turn generate some other input to one or more tasks.

For example, a mouse button may be pressed when the pointer is anywhere on the screen. This can cause one of the following to happen:

1. The task deals directly with the pointing.
2. The input focus is switched to the task's window and it then deals with the pointing.
3. The Shell deals with the pointing.

### 3.1.5.3 The Input Focus

The Input Focus is the place to which keyboard input is directed at any time. One window at a time has the Input Focus. The window which has the input focus is distinguished by:

- Being on top of all other windows.
- Having its window title showing selected emphasis.

Input focus can be changed either by a mouse Pointer selection in another window or by use of the "Switch Task" key, or by using the Task Manager.

### 3.1.5.4 Window manipulation - the System Menu.

The Shell provides a set of functions to allow the user to change the shape, size and position of screen windows. These functions are contained in the *System Menu*, which the user can access by selecting the System Menu icon (small icon on left side of the title bar) with the mouse, or pressing Shift+Esc. The System Menu contains the following functions:

- Restore
- Move
- Size
- Minimize
- Maximize
- Task Manager
- (optionally) Close

Applications can add **Close** to the System Menu if they wish to support double click on the System Menu as a fast path to **Close** an application. They must still support **Exit** on the application menu in this case. *Note:* for default VIO applications the System Menu will contain **Close**, and will also contain **Mark**, **Copy**, and **Paste**. See the section, "Copy and Paste for VIO Applications".

#### *3.1.5.4.1 Z-ordering*

In considering some of these functions, the concept of Z-ordering may be useful. It is notionally the third dimension of the screen and accounts for the order in which windows overlap each other. The top most window visible is the highest in the Z-order; the bottommost is the lowest. In terms of pieces of paper stacked on top of each other the Z-order is the depth and order of the pile. The Z-ordering also controls the jump ordering of applications. See the section, “Jump Ordering” for details.

#### *3.1.5.4.2 Window Maximize*

An application may define a size to appear when the user selects maximize. This size cannot be larger than the screen size, although neither window title nor borders need be shown if the application needs the maximum screen area.

To achieve this the user either clicks at the Maximize icon (with button 1) on the window title bar, or selects Maximize on the System Menu for that window. While the window is maximized, the Maximize icon on the title bar is replaced with the Restore icon. Maximized windows may be returned to their original size and screen position with Restore, or sized in the normal way. (If the window is resized, the Maximize icon is returned to the title bar and the Maximize command is reenabled.

The maximize key (Alt+F10) toggles. While the application is maximized it performs **Restore**.

Applications can be run with a smaller screen area, but may be maximized at the user's request. This smaller size might typically not cover the icon “Parking-Lot”. (See the section, “The Parking-Lot”)

#### *3.1.5.4.3 Window minimize*

In order to occupy as little screen area as possible, applications may be minimized. This will shrink the application to a predefined (iconic) bit-map.

To achieve this the user either clicks on the Minimize icon (with button 1) on the window title bar, or selects Minimize on the System Menu for that window. When the window is minimized, its appearance is defined by the application, but is normally a small bit-map giving a visual clue to the program function. When a window is minimized it is moved to the bottom of the z-ordering, and the next non-minimized window is made active. Minimized windows may be returned to their original size with Restore, or double clicking on the bit-map icon.

The minimize key (Alt+F9) toggles. It performs Restore while the application is minimized.

There are two possibilities for where a minimized window goes:

- to the place it was when last minimized. Minimized windows can be moved around the screen like other windows.
- to the icon “Parking-Lot”. if it was never minimized,

In either case, minimized windows are never overlapped. They are positioned on a notional grid on the screen, and if one position is occupied, the next position to the right, (then in row above) is used.

The position of minimized windows is not related to their position when restored.

#### *3.1.5.4.4 The Parking-Lot*

The icon “Parking-Lot” is this notional grid which overlays the screen. Each grid segment is large enough to contain exactly one icon; the grid starts at the bottom left of the screen, and goes from right to left, top to bottom. All icons will be aligned to this grid pattern.

#### *3.1.5.4.5 Change Window Size*

This is achieved from the mouse by pointing at the window border and selecting one of the four sides or one of the four corners.

- If a side is selected, that side may be moved towards or away from the opposite side. The opposite side is unchanged. The window becomes larger or smaller in one dimension only.
- If a corner is selected, the two adjacent sides may be adjusted to make the window larger or smaller in two dimensions at once.

In either case, the extent of the new window borders is indicated with an outline box which moves with the mouse. When the mouse button is released, the window occupies the position and extent indicated by the box.

From the keyboard, sizing is from the System Menu. Changing the size is then achieved by use of the arrow keys to move the corner or edge in the indicated direction.

The first up/down left/right arrow key hit will identify the horizontal and/or vertical edge to be moved.

This can result in the window borders being moved just off the screen and thus becoming not visible.

#### *3.1.5.4.6 Window Move*

To move a window with the mouse, the "press-and-hold" technique is used. The user points anywhere in the window title bar and then drags an outline box to where the window is to be positioned.

The window is redrawn when the mouse button is released.

Windows may be moved off of the screen to the extent that their title bar remains visible.

From the keyboard, the user selects "move" from the System Menu, and then moves the outline box using the cursor keys. The same restrictions on window position apply.

#### *3.1.5.4.7 Restore*

**Restore** returns the window to its last (unmaximized, unminimized) position and size.

Size or move operations between maximize or minimize do not affect this position. Thus the window can easily be returned to its "normal" position using **Restore**.

The user either clicks on the **Restore** icon on the window title bar, or selects **Restore** on the System Menu for that window. For mouse users double click on the title bar is a fast path for **Restore**.

The position of a window is only "remembered" when it is maximized or minimized from some intermediate position.

### **3.1.6 File Cabinet - functions for using Directories and Files**

The File Cabinet is a major feature of the system for most users. It is a program which lets users display and manipulate their file system, including any network connections. The file system is represented visually.

The File Cabinet provides a range of functions which can be performed on these items, such as opening a file (which creates an instance of the appropriate application for manipulating that file), moving a file into a directory, opening a program (which creates an instance of that program, without specifying a particular file to be worked on), copying a file, etc.

The File Cabinet and its associated windows are accessible to the user while running other applications. This allows the user to locate and browse other files at any time.

Working with files and directories in the File Cabinet would normally be through direct manipulation with the mouse. For example, to move a file from one directory to another, the user selects the file and drags it to the open destination directory.

Double clicking on an object, or selecting it and hitting Enter will open that object, which gives an object-oriented appearance to the system.

To unify the concepts of a file manager and application manager (for starting programs), a special “Startup” window has been created which contains directories and programs. Programs may be given long, descriptive names, yet are viewed and manipulated in the same way as the rest of the user’s file system. Information about the user’s file system is presented to the user in the form of three types of windows:

- The File Cabinet Window
- The Tree
- Directory Windows

Only one menu bar is available for the entire File Cabinet. This contains two sorts of functions:

- functions which operate globally on the entire file system
- functions which operate on the contents of the current directory

All directory windows are child windows of the File Cabinet window. In the File Cabinet, child windows are created slightly smaller than the window from which they came, and are slightly offset to the right from their parent. Thus if the Tree is sized to be half the area of the File Cabinet window, all subsequent directory windows will be slightly smaller than that size. If the File Cabinet window is minimized, all child windows temporarily disappear. If it is maximized, windows already created are not affected. If it is reduced in size, all child windows are clipped to the size of the File Cabinet.

Child windows may be sized moved, or maximized (but not minimized).

The title bar for these windows includes the name of the Drive/Directory (truncated if necessary).

One or more objects may be selected in the list using the normal selection mechanisms.



The differences between a Directory and Drive are relatively few, with some limitations on which actions the user may take on a Drive. For example, you cannot move a drive into a directory. Similarly, drives may appear based on an explicit action taken by the user to expand the file system - such as by connecting to a remote drive on a network. The things that can be done to the contents of Drives and Directories are the same, however, so the commands available are identical.

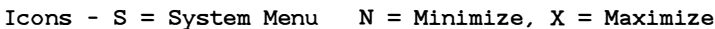
#### **3.1.6.1 The File Cabinet Window**

This window contains no data, but represents the maximum screen size to be taken up by filing system windows. It has an action bar which provides options for the topmost child window.

#### **3.1.6.2 Tree Window**

This window consists of a main area with a representation of the drives and directories in the system. It has no menu bar, and its size is limited by the bounds of the main File Cabinet window.

On the left of the main area, each drive in the system is represented by an icon in the shape of drive or diskette, accompanied by the disk volume name.



### Figure 3.1 The File Cabinet with Tree

and down keys. Hitting the right arrow key causes the next level of the tree to be revealed, and the first directory in that level to be selected. Hitting the left arrow key causes the current level of the tree to be hidden. Pressing alphabetic keys causes the selection to jump to the next directory that matches the key in that level; the next level of directory is displayed, as if right arrow had been hit.

The up/down arrows always remove any tree displayed at a deeper level.

*PgUp* and *PgDn* scroll the current directory level by a screenful. *Home* moves to the top of the current directory level, and *End* moves to the bottom.

As with the first-level directories, subsequent levels of the tree are linked visually with their parent directory or drive by lines or braces. Each column will have its own scroll buttons as required.

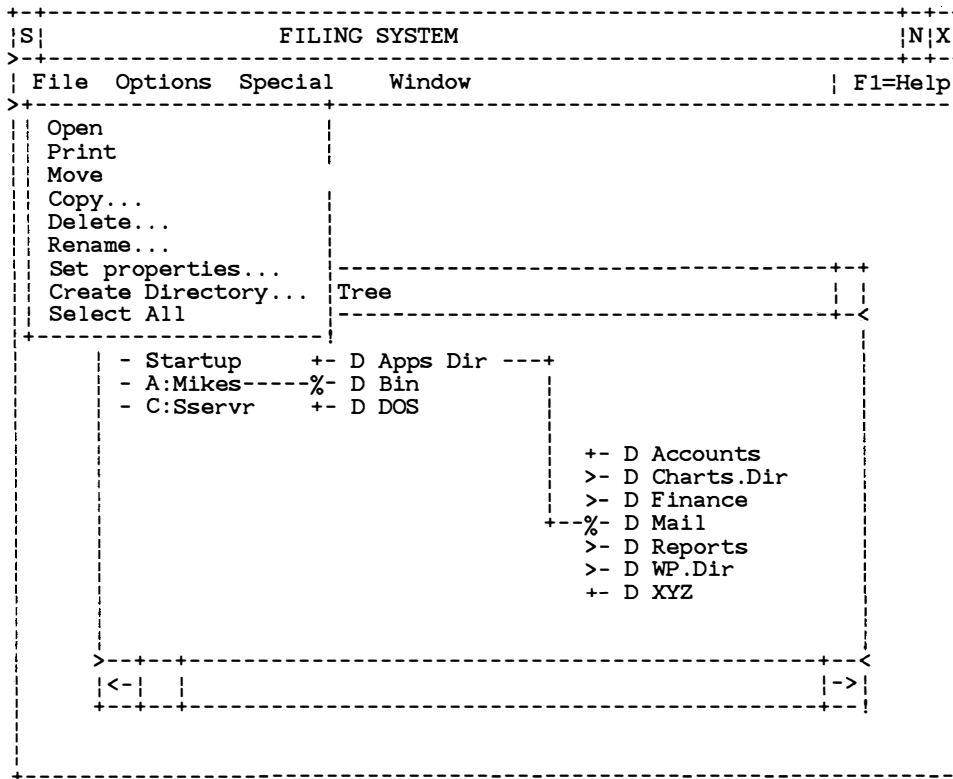
The directories may be nested so deeply that all the columns of directories cannot be shown in the window simultaneously. In this case, a horizontal scroll bar appears at the bottom of the directories window. This can be used to browse the whole of the directory structure.

No files are shown in the Tree window. The files and directories belonging

to a directory can be displayed in a directory window.

### 3.1.7 File Cabinet functions

#### 3.1.7.1 The File menu



Icons - S = System Menu N = Minimize, X = Maximize

**Figure 3.2 The File pull-down**

The File menu includes:

- 
- Open** opens a new window containing the selected drive or directory. The new window is displayed slightly offset from the Tree window, and on top of (first in the Z-ordering) all other windows. Windows showing directory contents are created based on the size of the window from which they were created. If a window containing the drive or directory

already exists, a new window is not created. Instead, the existing one is brought to the top.

The Open command is the default action for the File Cabinet window, hence double clicking an item, or hitting enter with an item selected will cause that item to be Opened.

- If the selected object is a directory, the new window is simply another directory window, containing the contents of the selected directory.
- If the object is a program, the new window contains another instance of the selected program.
- If the object is a file, and there is a default action assigned to the file then the default action is taken. (For example, .SCR might be input to a script program, when opening a .SCR file would run the script program with the particular .SCR file as an input parameter.)

If there are multiple default actions assigned to a file type (extension), an intermediate window appears with a list of the actions. The user is allowed to select one of the actions, the window disappears, and a new window belonging to the selected action appears.

Windows other than directories are not confined to the File Cabinet, but have a size and position determined by information stored with the program being invoked. If multiple items are selected when Open is requested, the shell assumes that the intent of the user is to use all of the items simultaneously. Thus all the items are opened. Any dialog boxes are shown sequentially.

**Print** causes the selected objects to be printed. If there is no program assigned to the object, then the object is printed as a text file. If there is a program assigned to the file extension, the program is invoked. Programs may provide a special invocation option for printing.

**Move** This command is provided for users who need to move files across drives and directories. It brings up a dialog box which contains two text entry fields. The first (From:) contains the names of the objects to be moved. When the dialog box first appears, this field is filled with all selected directories and files in the File Cabinet window. The user can then type additional names. Multiple filenames are allowed in the From: field, and standard wildcard syntax is valid. The second field (To:) contains the name of the destination. It is initially blank.

The user completes the move by typing in the name of the target file or directory and hitting Enter. Leaving the field blank implies the current directory (see Create Directory for

a description of the current directory).

If the user types into the From field, the typed text is added to the pre-filled text. The pre-filled text can be deleted.

It will be verified that there will be enough disk space before attempting the actual move or copy, so as not to risk running out midway through the operation.

**Copy** This command is identical to the Move command, except that it makes a second copy of the data.

Copy is also available by direct manipulation in a similar way to Move, but holding down the Alt key as well as the appropriate Move key(s)

**Delete** brings up a dialog box with a single text edit item, containing all currently selected items. The user can then type additional items. When the user confirms the delete, all items described in the text entry field are erased. Confirmation is by pressing Enter, or clicking on Delete.

Certain erase actions (drive contents, for example) cause further dialogs to confirm the operation, or ask for more details.

**Rename** This is used to change the name of a file, directory or file. Only a single file can be renamed.

**Set Properties**

allows the user to set the attributes of an object. The MS OS/2 file system attributes that may be changed are the **read-only** and/or **archive** bits, or the time and date information. Directories cannot be changed.

**Create directory**

This allows new directories to be created. The command brings up a dialog box to prompt the user for the name.

The directory is created in the current directory, which is the topmost window in the File Cabinet.

1. If the topmost window is the Tree, and only a drive is selected, it is the root directory of that drive. If it is showing some other directory selected, the current directory is the one shown selected.
2. If the topmost window in the File Cabinet is some other directory, then this directory represents the current directory in MS OS/2 terms.

**Select all**

Selects all objects in the current window. It is not valid in the Tree window.

### 3.1.7.2 Direct manipulation

Using the mouse, a completely different method of moving files and directories is available. The user selects an object to be moved, holds down mouse button 1 over the object, and drags it to a previously opened directory window. When the button is released, the object is moved to this new directory.

Both the source object and part of the target directory must be visible. Objects may be dropped anywhere within the target directory with the same effect.

Both source and target directories are redrawn after the object has been moved. The details of the move are as for the keyboard version, with the same error situations, but with an additional error when the target is not a valid directory window.

Multiple objects may be moved by extending the selection in the usual way, and then holding shift while dragging. Alternatively, if the space bar has been used to switch to extended select mode, then multiple objects will be dragged without the use of Shift.

Pointer appearance during the move is that of a file, or directory, or a group of objects for extended selection.

#### *3.1.7.2.1 Summary of Mouse use in Direct Manipulation.*

The following section details the rules for keyboard and mouse interaction for direct manipulation in the File Cabinet.

The default direct manipulation operation in Presentation Manager is a move on a single object. In order to perform a non-standard move operation or standard/non-standard copy operation some interaction is required with one or more of the following keys.

---

ALT key

This changes the operation from a move to a copy.

CTL key

This is used to add an object to a non-contiguous set for a group copy/move operation.

Shift key

This is used to add an object to a contiguous set for a group copy/move operation.

Below is a glossary of terms used in the following set of rules

- XS      Extended Selection Mode. This mode is entered by hitting the *<SPACEBAR>*
- Drag     Move mouse with button depressed for more than a predefined distance.

Shift + Click	Ctrl + Click	Click
Causes all objects from prior location to current location inclusively to be selected.	Causes this object to be added to the set of selected objects (removed if it was already selected).	Causes object to be selected mode it adds to the selection, otherwise anything else that was selected becomes deselected.

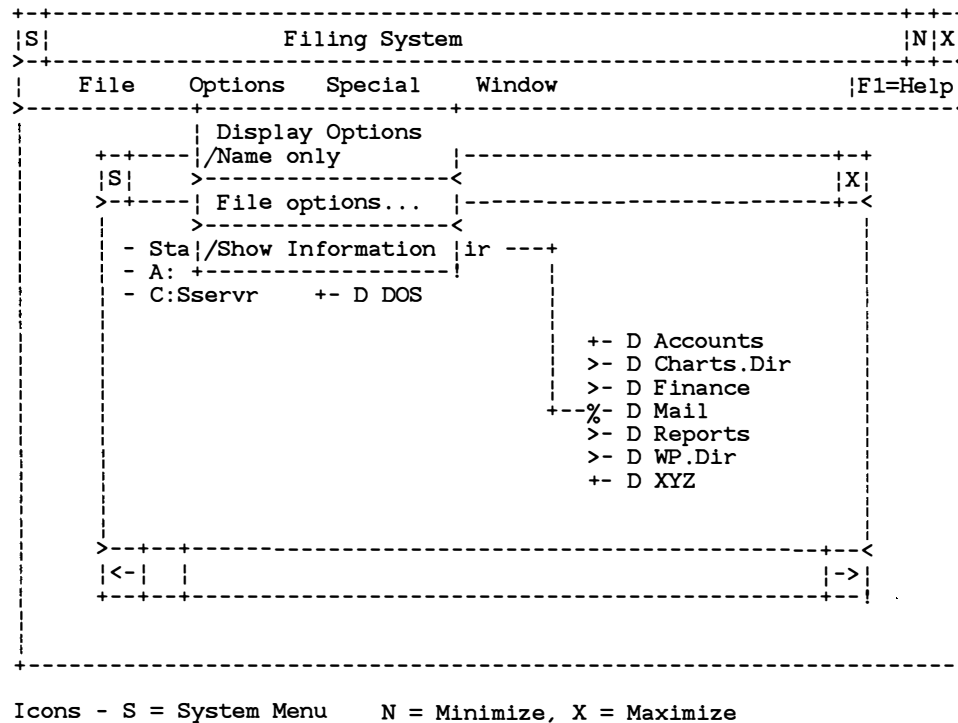
Figure 3.3 Key/Mouse Click Usages for Selection and Manipulation

DRAG					
Shift	ALT	CTL	Causes object to be selected and invokes the move		
Selects range of objects from prior cursor location to this object and invokes group move	ALT+Shift Selects range of objects from prior cursor location to this object and invokes group copy	Causes object to be selected and invokes copy. If in XS mode it adds object to set and invokes group copy	ALT + CTL Adds object to set of selected objects and invokes group copy	Adds object to set of selected objects and invokes move	Causes object to be selected and invokes the move in XS mode it adds object to set and invokes group move

Figure 3.4 Key/Mouse Drag Usages for Selection and Manipulation



### 3.1.7.3 Options menu



**Figure 3.5 The Filing system with Options pull-down**

The options menu applies to the active child window, and (optionally) to windows subsequently created in the filing system.

Windows previously created are not affected by changing these options.

#### Display Options...

The Display Options dialog includes

1. *Include*: which directories and files to display - the user can set this according to all valid attributes:
  - Name or extension - via wildcard filters
  - Type - directories, programs or files (check boxes).
  - File attributes - normal or special check boxes. Special shows a panel with hidden, read/only, system, archive check boxes. These selections show only objects with all the chosen bits set (and which qualify by the other selection criteria).

The objects displayed are the logical intersection of each of the groups selected. The defaults are \*.\* , all directories, files and programs, and normal.

2. *Display order*: what information the objects should be sorted on:

- Name
- Extension
- Type (directory, program, or file)
- Size
- Date/time

3. *What to display*: Name, date, size and attributes.

This allows the user to set the way that the contents of directories and drives are displayed.

#### Name only

An extra choice is provided on the first menu as a fast path to the display of as many objects as possible. When Name Only is selected, the other options for what to display are ignored, and only the names are shown. The choice toggles.

The name is always displayed. When “Name-only” is selected, objects are displayed in multiple columns, with a horizontal scroll bar appearing if there is insufficient window space to display all objects. Otherwise, the objects are displayed in a single column, possibly with a vertical scroll bar.

#### File options

toggles whether certain confirmation messages are displayed.

- Verify on Copy - compare the bytes in files after a copy
- Verify on Delete - show the dialog box on all delete commands
- Replace existing file - give warning prompt
- Sub-Tree Delete - delete a directory (and everything in it) even if the directory is not empty

#### Show Information

toggles whether information about the drive or directory is displayed. The state is indicated by the presence of a checkmark (on) or its absence (off). If off, no information is displayed. If on, the following is displayed:

- Space used by files shown (files that have passed filter), out of total on disk.

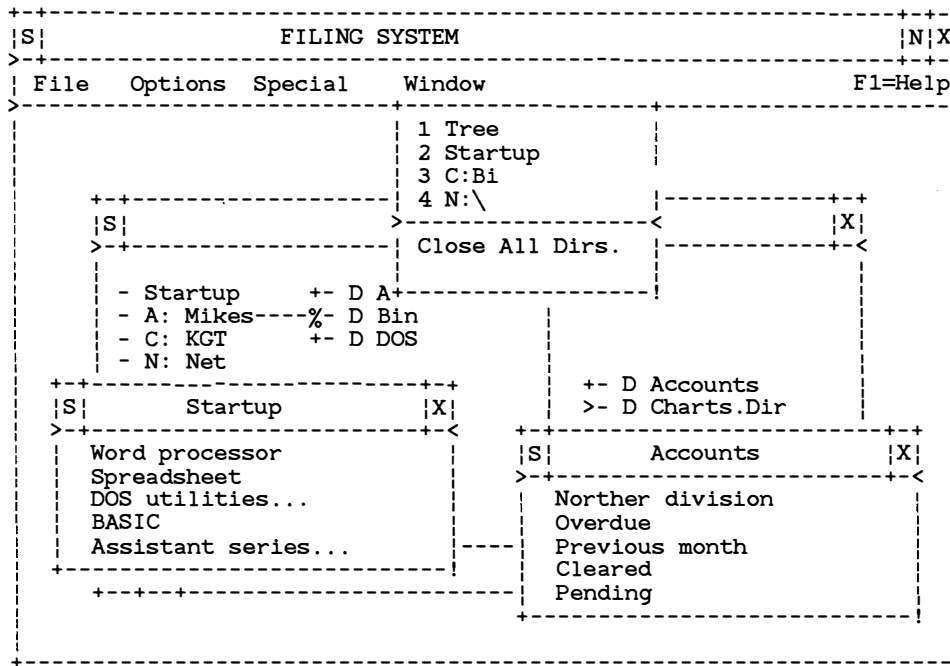


- format single/double density.

- Format prompts the user for a volume label and other

Refresh Ensures that the File Cabinet windows are all up to date.

### 3.1.7.5 The Window menu



Icons - S = System Menu    N = Minimize, X = Maximize

**Figure 3.7 The File Cabinet with Window pull-down**

- F6** The next directory in the currently displayed directories is brought to the top of the child windows and given the input focus when F6 is hit.
- Directories are removed by closing them using their System Menu/icon.
- 1 Tree** The Tree window is given the input focus and brought to the top of the child windows in the File Cabinet
- Directory list 2..n**  
A list showing the child windows in the File Cabinet is shown in the pull-down window. If there are more than 8 directories, an option to list all the directories replaces the ninth.
- The name shown is the full name of the directory displayed.

For the root directory it is the drive identifier and backslash.

The names of Startup program directories are truncated to 40 characters.

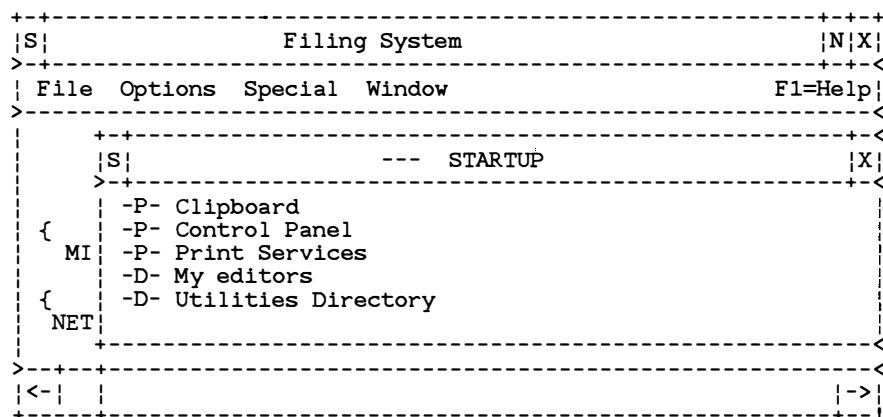
#### Close All Directories

This commands closes all existing directory and drive windows, providing an easy way for the user to clean up the File Cabinet windows.

### 3.1.7.6 STARTUP window

The Start-A-Program functionality of MS OS/2 is represented by a special STARTUP icon which appears in the Tree window along with the other Drives. Its functions and operation are essentially the same as any other directory window, with some exceptions:

- Directories are represented by program **Groups**.
- Activities must be added to STARTUP using a special application called the “STARTUP-Editor”. This includes the ability to install an activity into the list, or modify an installed entry. This file is available in the File Cabinet by selecting the “STARTUP Editor” from the list of programs in STARTUP. Applications may also add themselves to STARTUP during installation.
- Entries in the STARTUP list can only be moved within STARTUP.



Icons - S = System Menu      N = Minimize, X = Maximize

**Figure 3.8 The File Cabinet with STARTUP panel**

#### 3.1.7.6.1 *STARTUP Functions*

The File menu functions are:

---

Open	Causes selected STARTUP program to be run, or a group to be opened.
Copy	Works only within STARTUP, but is otherwise similar to normal Copy. Only one program or group may be copied at a time using this method. Names may be fully qualified using “\”  Direct manipulation works with the STARTUP programs and groups, and does allow multiple items to be moved and copied.
Move	Used only to move between groups in STARTUP. Similar to Copy.
Rename	Used to change the long name of a program.
Delete	Deletes the reference to an application from the directory. A warning is given if this is the last reference to an executable file.
Create directory	Creates a new group of applications in STARTUP
Select all	Selects all activities in a group.

In the other pull-downs, File Options is available, plus all the Window

options. All other options in pull-downs are grayed.

In the Tree, application names are displayed as the full length of the text (no trailing or leading blanks). The column widths are adjusted to support the longest entry in that group.

In windows showing the contents of program groups, the names of programs and groups are displayed in a single column. No other information is shown, apart from the directory/program icon.

### 3.1.8 STARTUP Editor

Most programs in Presentation Manager are added to the system using the standard installation process. The STARTUP Editor is only needed for those programs not installed in this way.

The STARTUP Editor is available from the File Cabinet, and allows users to add or modify an entry in STARTUP. The STARTUP Editor consists of a main window which includes entry fields that the user enters the relevant program invocation data, including:

- Program type (Radio button choice)
- Icon file name.
- Executable file name and directory
- Working Directory at invocation
- Parameters. This will have a syntax which allows prompting.
- Two-line description of the application (also provided during installation)
- Environment variables
- Program name for STARTUP
- Group in STARTUP

The last two items are prompted for by Save As...

The information supplied is stored in the PRESSERV.INI file.

All of the information which can be edited here is given initial values automatically during installation.

```

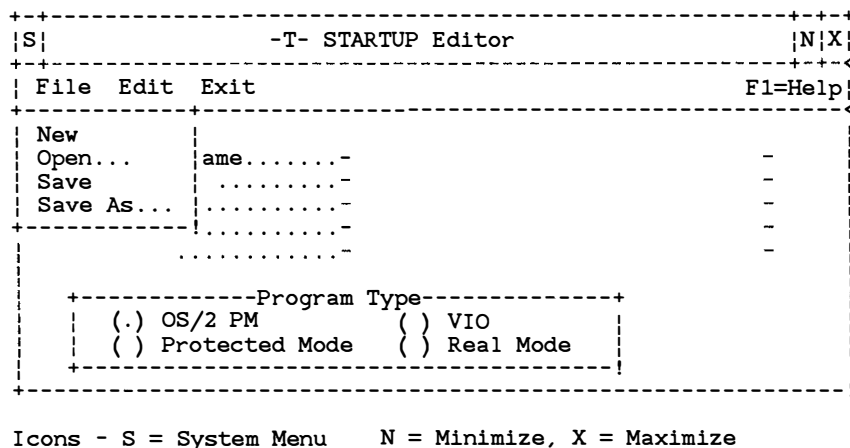
+-----+-----+
|S|          -T- STARTUP Editor          |N|X|
+-----+-----+
| File  Edit Exit                          F1=Help|
+-----+-----+
|
| Path/Program name...-                    -
| Icon file name.....-                    -
| Parameters.....-                        -
| Working Directory...-                    -
| Environment.....-                      -
| Description for Help-
|
| +-----+-----+
| |          Program Type          |
| | (.) OS/2 PM          ( ) VIO   |
| | ( ) Protected Mode   ( ) Real Mode |
| |          +-----+-----+
|
+-----+-----+

```

Icons - S = System Menu    N = Minimize, X = Maximize

**Figure 3.9 Startup Editor - main panel**

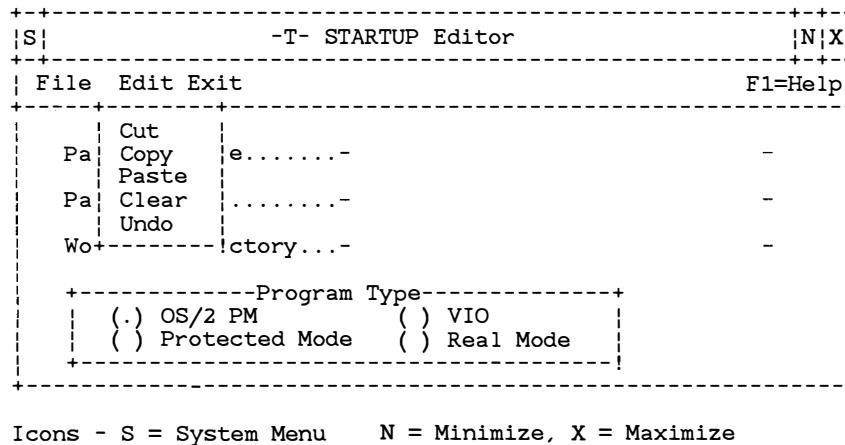




**Figure 3.10 Startup Editor - File pull down**

The File menu includes:

New	This command clears the any entries in the STARTUP Editor window and resets any default fields.
Open	This command is used to load a previously defined STARTUP entry. A dialog is displayed which allows the user to selected the entry from a list box.
Save	This command is used to save a STARTUP entry. If a name has not been supplied a dialog is displayed which prompts for a name, otherwise the information is simply saved. The group name is prompted for in a similar way.
Save As	This command is used to save a STARTUP entry with a given name. A dialog is displayed with an entry field which prompts for the name. If the program already has a name, it is proposed as the default. The group name is prompted for in a similar way.



**Figure 3.11 Startup Editor - Edit pull down**

The Edit menu includes:

Cut	Removes the currently selected text and places it on the Clipboard.
Copy	Places a copy of the currently selected text on the Clipboard.
Paste	Inserts the current text contents of the Clipboard in the selected field.
Clear	Removes the currently selected text, but does not place it on the Clipboard.
Undo	Regress the last change.

### 3.1.8.1 The Exit menu

This command quits the STARTUP Editor. If any changes have been made to the entry's information since the last save, a message will be displayed informing the user that there are unsaved changes and requesting whether to save changes. If the user responds Yes (the default), then the Save As dialog box is displayed, if not, then the STARTUP Editor ends.

### 3.1.9 Task Manager

Presentation Manager is capable of running many tasks at the same time. These programs can all potentially be using the screen at the same time. The user needs to be able to identify which tasks are running and to control which tasks are visible on the screen at a given time. The user also needs to close the system down at the end of work.

#### 3.1.9.1 How to Access The Task Manager.

The Task Manager is a window, which appears in the Presentation Manager screen group, that contains a list of the user's current activities. The user brings the Task Manager window into view by hitting button 3 on the mouse, hitting Ctrl+Esc on the keyboard, or by selecting the Task Manager command from the System Menu of the current application. If the Task Manager is called up while the user is in a non-Presentation Manager screen group, an implicit screen group switch is performed, and the user sees the Task Manager in an otherwise empty workspace.

The Task Manager normally contains a representation of every independent task running in the system. The representation is in text form, where the text is provided by the object, and typically matches the text displayed in the caption of the object (file name + data name).

The default Task Manager window size is large enough to display approximately 10 objects. A vertical scroll bar is displayed and can be used to move the list of objects, allowing every object in the Task Manager to be viewed, even if not all objects fit within the window. The object last worked on is selected and the list is scrolled to show the selected item at the middle of the window (unless the first or last item is visible in the window).

The appearance of the Task Manager window with several tasks running might be as follows:

+-----+   S   TASK MANAGER +-----+		
Control Shutdown		F1=Help
+-----+		
ALPHA.EXE		A
Clipboard		---
Control Panel		---
My Diary		
Notepad - (TEXT.TXT)		
Spreadsheet (ACCTS.SPD)		
Paint program (DIAG.DOC)		---
XYWRITE.EXE		V
+-----+		

**Figure 3.12 The Task Manager window**

The objects in the Task Manager are ordered alphabetically.

The entries in the Object List are selectable.

The System Menu commands include:

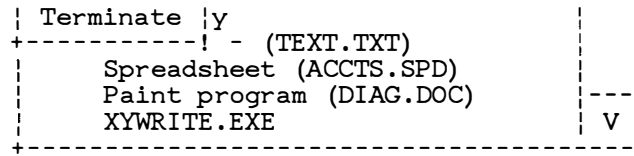
- Switch to
- Close (Same as Exit being selected in application)
- Terminate

### 3.1.9.2 Jump Ordering

The jump order round applications is their Z-order on the screen. Applications can optionally not participate in the jump sequence, but normally all applications will participate.

Non-Presentation Manager applications appear as icons in the Presentation Manager desktop, and so have a logical entry in the jump order sequence. Using the keyboard or mouse to jump to the next application will make the icon representing the non-Presentation Manager program active. To see the program itself the icon will have to be **Opened**. Note that if the corresponding entry in the Task Manager window is selected that the icon is automatically opened.

+-----+   S   TASK MANAGER +-----+		
Control Shutdown		F1=Help
+-----+		
Switch To	XE	A
Close	rd	---
< Panel		---
+-----+		



**Figure 3.13 The Task Manager window with Control pull down**

### 3.1.9.3 How to Work With a Task.

The user can choose to work with a particular task by:

1. Selecting the Name of the task in the list.
2. Select the Switch To command on the Control menu.

The fast way of getting a task to be the Active one is to do a doubleclick on it with button 1 or select it and press Enter.

The selected task becomes the Interactive Program.

Certain tasks can cause the Switch To selection to be grayed.

Alphanumeric keys can be used to move the selection to an object on the list when the first letter of the name matches the key pressed. If there is more than one match the selection moves to the first. If the same key is pressed again, the selection moves to the next and so on, recycling at the end of the matching section to the top. If no match is found, the machine beeps.

For *Presentation Manager* tasks, this causes the main window of the object to appear on top of the other windows in the Presentation Manager screen group. The application may choose to bring its other windows to the front at the same time. Keyboard input is directed to one of the windows belonging to the task.

For *Non-Presentation Manager* programs, the Presentation Manager Screen Group is removed from the display and is replaced by the Screen Group containing the program. This occupies the whole screen and no other programs can be seen. Both Mouse and Keyboard input are directed to the program.

Non-Presentation Manager programs also show up as icons in the Presentation Manager screen group.

### 3.1.9.4 How to Close a Task

Some programs will not have a Close(Exit) command. The user may use the Close command to close these objects. This command requests that the program close down normally (save data, clean up).

### 3.1.9.5 How to Terminate a Task.

Most programs have their own methods of being brought to an end normally, through menu options or commands. Generally the user should use these methods to terminate a running program. This is advisable because the program probably needs to tidy things up before it finishes - save work away in files, for example.

However, it can happen that a program gets stuck or begins to behave in an unusual way. To allow the user to stop such a program, the *Terminate* function can also be used to quit a program. When invoked the Terminate function causes a destructive shutdown, i.e., the program does not get a chance to save data or otherwise clean up.

To terminate a program in this way, select the program's entry in the Task List and then select the *Control* option in the Task Manager menu bar. Select the *Terminate* option on the pull-down menu which appears as shown in. A warning panel is displayed. This allows the user a second chance to think about the destructiveness of the Terminate function and prevents inadvertent program stopping.

The filing system window cannot be closed or terminated.

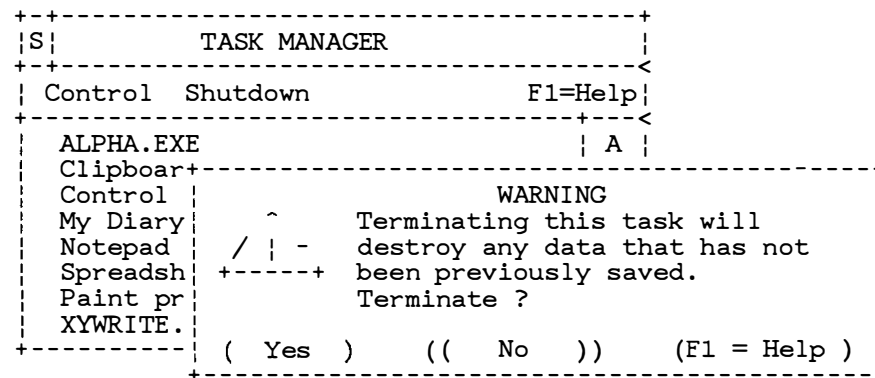


Figure 3.14 Task Manager - terminating a task

Either the Yes or No buttons must be selected. The default button is No (for safety), in case the user presses the Enter key as the first action after this panel appears.

Selecting the *No* option quits the Stop operation and leaves the program running.

Selecting the *Yes* option causes the program to be stopped.

+-----+   S   TASK MANAGER +-----+<			
Control Shutdown		F1=Help	
+-----+<			
ALPHA.	Shutdown now		A
Clipbo	/Save at shutdown		---
Contro	Save tasks now		---
My Dia			
Notepa			
+-----+<			
Spreadsheet (ACCTS.SPD)			
Paint program (DIAG.DOC)			
XYWRITE.EXE			
+-----+<			

Figure 3.15 Task Manager with Shutdown pull down

The Shutdown menu includes:

Shutdown now

This is the normal way to close the system down at the end of the work session. There are two variations, controlled by the next option in the menu.

1. **Save at shutdown on** causes Shutdown-now to save the entire task list in terms of the applications running, and their position on the screen. Each application is responsible for saving its data and current state.
2. Shutdown with **Save at shutdown off** causes all applications to end normally but no record is kept and Restarting the system will not restart the current set of applications. Before an application ends, it is expected to prompt the user to save any unsaved changes, and then to shut down.

Save at shutdown

This toggle indicates the current action to be performed at shutdown. It is initially set off (no save).

**Save tasks now**

This provides the user with an easy way to save the layout and data ready for the next IPL. This includes notifying applications so that they can be restored at least working on the same file at the next IPL. Screen window layout and current options must also be preserved.

No shutdown is performed.

The Task Manager window is removed from the screen only following a Switch To operation.

**3.1.10 Control Panel**

There are many options the user has for how the system works. Most have to do with the hardware configuration, while others have to do with the Presentation Manager system's appearance. Control Panel allows the user to change these settings:

```

+-----+-----+-----+
|S|               Control panel               |N|X|
+-----+-----+-----+
| Preferences Settings Configuration           |F1=Help|
+-----+-----+-----+
|
|   Presentation Manager Version ...
|
|           +-+       +-+
|         Date +-!   Time +-!
|
|   Cursor blink  |-----|
|   Double click  |-----|
|
+-----+-----+-----+

```

Icons - S = System Menu      N = Minimize, X = Maximize

**Figure 3.16 Control Panel**

**3.1.10.1 Main panel****Time & Date**

The user can set the time and date (Entry fields). This will set the internal hardware clock.



**Double click**

The double click rate is the time interval within which individual mouse clicks must be received in order to generate a doubleclick.

**Cursor Blink:**

This changes the rate at which the caret flashes. Cursors which flash too quickly tend to be distracting, while if they are too slow, then they are hard to spot.

**3.1.10.2 Preferences pull down**

---

**Sound On or Off**

Allows the user to turn off sound (Check box)

**Screen Colors**

The user can select which colors are used in various parts of the system. For example, the items below can be changed. This list is not considered to be exhaustive.

- Window Background
- Window Text
- Scroll Bar
- Scroll Arrows
- Scroll Elevator
- Active Title Bar
- Inactive Title Bar
- Title Bar Text
- Window Borders
- Menu Bar
- Menu Text
- Screen Background

**Border sizes**

The user is able to select border width. Novice users may want wider borders to make it easier to manipulate the borders directly. More experienced users may want to shrink the borders.

**Logo on/off**

The user can suppress all logo displays. Default is no suppression. (Check box).

#### Mouse Buttons

Right handed people tend to want their mouse buttons from left to right (1-2-3), whereas left handed people want the opposite (3-2-1). Control Panel allows the user to select which he or she wants.

#### 3.1.10.3 Settings pull down

---

##### Printer drivers

Options to select additional printer drivers (Entry field)

##### Printer defaults

Options to select default printer and settings (List boxes)

##### Print spooler

Options to select spooler (Entry field)

##### Communications

Options to select Baud rates etc. (Entry fields, radio buttons)

##### Ports

Options change Comm1 etc.(List boxes)

For further information see the section, "Spooler and Printer Configuration" in the chapter, "Spooler Interface".

#### 3.1.10.4 Configuration pull down

---

##### Global data.

In the Presentation Manager system, "PRESSERV.INI" is not a text file, and if the user's system configuration changes, it is not possible for the user to edit the changes into "PRESSERV.INI" with a text editor. Therefore, the Control Panel must account for all such possible hardware changes. This includes changes to CONFIG.SYS. This includes changing the type ahead buffer size, and the autorepeat rate of the keyboard (this is configurable on the AT). The more commonly changed options are shown individually with entry fields, others are shown in list boxes.

For full details of the contents of CONFIG.SYS and an explanation of what each entry means the user should consult the MS OS/2 Setup Guide.

Path: The user can set the initial path for the system.

Fonts: It is possible to purchase new fonts for Presentation Manager and to tell Presentation Manager that they exist.

**International Settings:**

The user can change the time/date format, the currency symbol, and other international characters.

**Default action definitions.**

The default Open action and Print action may be specified according to file extension. The panel allows this for each file extension the user wishes to define, plus a default to be used by others, or the option to disallow certain file extensions.

When applications are installed, they may wish to prompt the user as to whether they should appear for certain file extensions.

List boxes and entry fields allow this.

**Miscellaneous system variables:**

Users can change any system variable through the control panel. Applications can define variables in the initialization file themselves, and list boxes and entry fields will be available to change these.

### **3.1.11 Clipboard Viewer**

Presentation Manager provides copying and moving using the cut, copy and paste metaphor. This allows an object/action approach to be applied to copying, rather than the action/object approach of "copy what, to where".

To the end user, the appearance of cut and paste is as follows:

1. Mark the object to be copied or moved by the normal selection process.
2. Choose either cut (to remove it from the file to the clipboard), or copy (to copy it). In entry fields, cut may be performed using the Delete key, and Copy using Ctrl+Insert.
3. In the target file (which may also be the original file), select the target position.
4. Select paste. In entry fields, Paste may be performed using Shift+Insert. This replaces selected text with the clipboard contents.

Typically, these functions are provided on an Edit menu in each application (As in the STARTUP Editor), and the user can use them both inside and between applications.

The Clipboard viewer provides the user with an easy way to look at the contents of the clipboard. As objects are cut and copied to the clipboard, the Clipboard Viewer displays the object type and the object itself. The Clipboard Viewer understands several predefined formats:

- Text
- Rich Text
- Bitmaps
- Metafiles

The Clipboard Viewer menu contains:

---

Clear	Allows the user to empty the clipboard and free up any memory used by the objects in the clipboard.
-------	---

#### **3.1.11.1 Clipboard mechanics.**

To create this easy-to-use environment, the clipboard is set up to accept any number of different data formats, several of which it can hold simultaneously.

When a new application wants to copy the data, it looks at the data formats available, and chooses one that it understands.

The data itself may be in the clipboard already, or it may be sent as the result of a message from the target application to the source one.

Applications should use standard IBM data stream definitions, since these will be supported by printers, plotters and editors.

#### **3.1.11.2 Copy and paste for VIO applications**

A simple form of cut and paste is provided for default VIO applications.

---

Copy	Areas of screen can be MARKed with an option available from the System Menu, and the contents of the marked area copied to the clipboard in text format.
Paste	will replay text as if it were being keyed into the application.

### **3.1.12 Initialization**

#### **3.1.12.1 The initial view of the system**

The view of the system when it is IPLed for the first time is that of the File Cabinet with the root directory of the default drive displayed in the Tree.

Also displayed is an open directory showing the root level of STARTUP.

This view will remain the initial view until it is replaced by the user saving the current tasks.

#### **3.1.12.2 The Initialization File**

The initialization file, called PRESSERV.INI, is a Binary file - i.e. it is not human readable and cannot be changed easily using text editors. The file is hidden, since it is not intended for direct use by end-users. Its use is mainly by the Presentation Manager system, which does provide some API functions for reading and changing its contents. (See the section, "Presentation Manager Initialization File Functions")

The initialization file contains all non-volatile task information. This includes files installed in STARTUP, open directories and running files, and system defaults for both MS OS/2 and Presentation Manager.

The system will boot without the initialization file, but default system settings for colors, display device etc., will be used.

If errors are detected within PRESSERV.INI, the system will attempt to function correctly, but some strange behaviour may be noticed. For example, applications that the user knows have been installed not known to STARTUP. The user will be informed of the errors, whereupon it might be necessary to restore the system from a backup copy.

On network systems there may be a local copy of PRESSERV.INI but a global copy of Presentation Manager.

### **3.1.13 HELP facility for the shell.**

The purpose of Help is to provide information to the user which aids in the operation of the shell. When the user requests Help, information regarding the item selected in the current context is displayed. The user can also request an index of available Help topics, request General Help, or request information on the functions assigned to keys.

### **3.1.13.1 INVOKING HELP**

The user can request Help by either pressing the F1 key, or by clicking the mouse pointer while it is on the F1=Help choice on the Action Bar. After doing one of these actions, a secondary window will be displayed, and it will contain a panel of information which pertains to the item on which the selection cursor is currently positioned.

Note that more than one item may be selected on the panel, but the initial help will relate to the item where the selection cursor is situated (generally the last item selected).

### **3.1.13.2 THE HELP WINDOW**

The Help window is moveable and sizeable, and will be the topmost window when it is active. It contains four pushbuttons along its bottom, and these make up the Common Actions Area. There is also a vertical scroll bar along the right side of the window, and this can be used to scroll the Help panel which is displayed in the window if it is too big to fit. If the window is sized smaller than its default size in the horizontal direction then the text is clipped, there is no horizontal scroll bar. The window must be resized to display the full text.

In the window's title bar appears the application name. Here is a picture of a typical Help window:

Filing System		N	X
File	Options	Special	Window
			F1=Help
Open			A
Print			
Move			
Copy			
Del			
Ren	Filing System Help		
Set			
Cre	Copy		A
Sel			
1/ Select the file to be copied in its directory More than one file may be selected by means of extended selection.			
2/ Select Copy on the File pull-down.			
3/ Type in the name of the target directory, using the directory index as a reference.			
4/ Select Copy on the Copy menu.			
(Esc=Cancel) (F1=General Help) (F5=Index) (F9=Keys)			
			V
			%
			->

Icons - S = System Menu    N = Minimize, X = Maximize

**Figure 3.17 A sample help window**

### 3.1.13.3 HELP INTERACTIONS

When a help panel is displayed in the Help window, the user can either use the mouse to scroll the text with the scroll bar, or can use the arrow keys, Home, End, PgUp, PgDn, Ctrl+Home, and Ctrl+End keys to scroll the text.

The pushbuttons at the bottom of the window, in the Common Actions Area, perform the functions described below. The user can perform the actions by either pressing the pushbutton with the mouse pointer, or typing the key whose label is on the desired button, ie. F1, F5, F9, Esc.

#### *3.1.13.3.1 F1=General Help*

Displays a general Help panel describing what the panel is for and the concepts behind it. This is not help on the Help facility.

#### *3.1.13.3.2 F5=Index*

Displays a selection panel (or listbox for simple applications) which lists all of the available Help topics. The user can select a topic from the list, and it will be displayed. When the listbox is displayed, the F5=Index pushbutton changes to read Enter. The user can select a topic by either pressing the Enter key, pushing the Enter button, or doubleclicking the mouse on a topic in the listbox. When the listbox is removed, the Enter pushbutton is changed back to read F5=Index.

#### *3.1.13.3.3 Shell Help Index*

For each of the Presentation Manager Shell “Applications”

- File System
- Task Manager
- Control Panel
- StartUp Editor
- Clipboard Viewer
- Printing Services

the HELP index will contain the following -

- A list of all the available HELP topics for the application
- An item to select help on the System Menu
- An item to select help on general controls (E.g. Scroll bars )
- An item that states that general help is available from the File Cabinet Help Index

In the File Cabinet as well as the above items there will be a list of Global Topics, e.g. Setting screen colors, Printing files.... which will contain a brief description of how these things can be done and say which application needs to be started.



#### *3.1.13.3.4 F9=Keys*

Displays a Help panel describing the functionality of all the keys available to the application (without Help displayed).

If the user asks for help on keys while in an application, the help will display the key functions for that application. The user will be advised that the application must be the interactive window before the keys will work as defined in the panel.

#### *3.1.13.3.5 Esc=Cancel*

Removes the currently displayed help window.

### **3.1.13.4 Additional notes about Help**

If there is no selection cursor when Help is requested, a general Help panel providing information about the currently active window will be displayed.

When the Help window is displayed, it becomes the interactive window.

During pull-down interaction with the mouse, Help must be selected using F1.

The Help panel exists until it is removed by the user, or its containing application is closed.

If the application is minimized while Help is being displayed, the Help panel is removed. If the Help panel is required when the application is restored, F1 must be pressed again.

Help is available by hitting F1 while a Window is minimized and has the input focus. Help for the System Menu will be displayed which will contain help on how to restore the window.

Help panels are not constrained within the application main window.

Examples will be included on Help panels where appropriate.

### **3.1.13.5 Help on items in STARTUP**

A two-line description is supplied with applications, and automatically included at installation. It may be edited using the STARTUP Editor. This brief application help will be displayed if **Help** is requested in the File Cabinet while an application is the selected item.

No index is provided for this Help.

## 3.2 The User Interface Shell API

*All* of the User Interface Shell functions that are available to the end user, through interaction with the Shell using the mouse and keyboard, are also available to an application program using the User Interface Shell API.

The API functions provided are listed and described below. The lists of functions are divided into functional areas that are similar to those for the description of the interactive components of the Shell given above.

### 3.2.1 Definitions of terms used in the Shell API

The Shell API introduces a number of unique terms and concepts which require definition, and those definitions are collected together in the following list.

**Application Information File** This file, usually provided as part of an application package, provides a simple way for a programmer to describe his “program” in a way that is acceptable to the Presentation Manager system. The file is an ASCII text file, which can be edited using any of a number of available Line- and Full-screen-editors, including EDLIN.

The file contains a (relatively) free format description of an executable file, or files, the title required for the program, a description of its parameters and so on.

The Shell API provides a function, WSHLoadAIF, which can read these files and convert the contents into a fixed format data structure in memory. This data structure is used as input to the WSHAddProgram and WSHChangeProgram functions in order to create or change the program’s entry in the Installed Program List.

The AIF file format is as described in the MS OS/2 Reference with regard to the PIP language. If a user wants to create an AIF and use that as a program definition file, Presentation Manager will use the AIF as long as it follows the definition laid down by MS OS/2.

**Executable File** The *fully qualified* name of the file that is executed when this program is run.

Executable files can be “.BAT” files, “.CMD” files, “.COM” files or “.EXE” files. Note that Presentation Manager can only start protect mode programs and will not be able to start real mode programs.

It is permitted for several entries in the Installed Program List to refer to the same executable file. This might be done if the same program is to be used with different (fixed) parameters at different times.

**Group Information Structure** This is a data structure which is used to return information to the caller of the WSHQueryProgramTitles function.

The format of the Group Information Structure is defined in the description of the WSHQueryProgramTitles function. **Program Information Block** This is a data structure which is used to contain a representation of all of the information stored in the Installed Program List about a single program.

The data structure is used to return information from the WSHQueryDefinition function, and to provide information to the WSHAddProgram and WSHChangeProgram functions. The format of the Program Information Block is defined in the description of the WSHAddProgram function.

**Handle** In general within Presentation Manager, a **Handle** is a 32 bit value. It provides an identifier to an object or resource being used by the owner of the handle.

**Icon File Name** When a program is started, the Icon that is to be used to represent that program is loaded using the resource handling functions of Presentation Manager.

The Icon loaded appears in the displayed representation of the Switch List.

**Icon Handle** When an Icon is loaded using the WINLoadIcon function, the function returns a handle that can be used to refer to that Icon.

The Icon Handle is stored as part of the Switch List entry for a program in order that the visual switch list program can display it.

**Installed Program List** This is a data structure maintained by the Shell API which contains the description of all "Programs" that the Shell can recognize as programs.

The information stored for each program includes the name of the "Executable File" associated with the program, a readable "Program Title" for it, and some execution control attributes for the program - "Program Type", "Program Handle", "Visibility", and a description of "Program Parameters".

**Presentation Manager Initialization File** In order to preserve system control information, for example, the "Installed Program List", all of the information held by the Shell API is actually held on disk. The information is stored in a single file, the Presentation Manager Initialization File.

The file is a binary file, the actual format of the data stored in the file is not published. Facilities are provided for applications to record their own permanent information as part of this file.

**Program** The smallest executable unit recognized by the Shell. A **program** will minimally include a **Program Title** (q.v.) and a reference to an **Executable file**.

As far as the Shell API is concerned, a program exists only after an entry for it has been added to the “Installed Program List” by means of the WSHAddProgram function.

**Program Groups** These are provided to allow the collection of an arbitrary set of programs into a unit that can be acted upon as a single entity.

The relationship of program groups to programs is similar to the relationship of subdirectories to files as given in the MS OS/2 file system. It can be useful to think of the program group hierarchy in the same way as you would think of the file system tree structure. Where the analogy breaks down, this specification will make the breakdown explicit.

Program groups appear in the Installed Program List as if they were themselves programs, and they have their own “program titles”, “program handles” and “visibility” attribute but none of the other attributes apply to groups. A program group can appear within another program group, i.e. the group structure allows nesting, but recursive group definitions are not permitted.

There are several reserved handles which refer to a reserved program group, the complete list of programs, and two groups that are concerned with system configuration and start up parameters. These handles always exist and cannot be destroyed. They are the “Root-Group”, “Master-List”, the “Auto-Load-Group”, and the “Save-Group”. At Program installation, a group handle must be specified and the program is added to the specified group and automatically inserted into the Master-List.

The same entry may appear multiple times within one group. This will have the effect of starting the same program multiple times if the option to start the whole group is selected. Note: the file system does not behave in this way, in that within any one subdirectory you cannot have duplicate filenames, where a filename would correspond to the handle for a program. Master-List This is similar to a program group, and can be read using a reserved handle, but there are several differences with respect to a “normal” program group. The **Master-List** can be thought of as a copy of the Installed Program List, except that its contents are available to a program. The Installed Program List is NOT made available.

All programs in the Installed Program List are automatically included in the Master-List, and cannot be removed unless the program definition is removed from the Installed Program List.

The Master-List is provided to “anchor” program entries so that the WSHQueryProgramTitles function can be used to find all the defined (meaning installed) programs.

There are NO group handles within the **Master-List**. All the entries are program handles. Further, there are no duplicate handles within the **Master-List** although duplicate titles may appear.

The value of the reserved handle will change if the format of group and program handles is changed, however it will always be the value of the target data type with all bits set to one. With the current definition of a group handle the value is 0xFFFFFFFF. The “Name” of this structure is “WinMASTER-LIST”.

**Program Handle** This is a number, assigned at the time a “program” is added to the “Installed Program List”, that uniquely identifies the program.

The number used is fixed from the time a program is added to the list until the time it is removed. The number assigned will not be used for any other program or “group”. The number that represents a particular program will not change across a system IPL.

Most Shell API function calls that reference programs use the program handle to identify a program; the program handle is the only unique reference to a program that exists.

**Program Parameters** When programs are invoked using shell they can be supplied with run-time parameters by the Presentation Manager system.

These parameter values can be fixed, or automatic interactive prompting for the required information can be provided. If interactive prompts are selected, a number of editing options are available to control the values supplied by the end user at run time.

The parameter generation is driven by information supplied at the time the program is installed, this information is stored as part of the Installed Program List entry.

**Program Reference** A program group is used to collect arbitrary programs together into a single unit. The collection process has no effect on the program definitions involved, the group merely indicates which programs it contains.

The “Program Reference” is, therefore, just a pointer which links a group to each of the programs defined in that group.

**Program Title** A *readable* name, usually in the end users natural language, which can be displayed to the end user.

The title allows the user to recognize programs more easily than using an executable file name, and allows tailoring to national languages to be simplified.

Program titles need not be unique; it is possibly, even likely, that several entries in the “Installed Program List” will have the same title, with the restriction that you cannot have duplicate titles within the same program group.

Program Titles are not allowed to be null. If a null title is passed when a program is added to the system the system will generate a title from the name of the executable file associated with that program.

**Program Type** Programs are divided into types based on their run-time attributes. They can be “Real Mode” programs, “non-Presentation Manager” programs that must be run in a new screen group, “non-Presentation Manager” programs that can run in the Presentation Manager screen group, or “full” Presentation Manager programs that *must* be run in the Presentation Manager screen group. These four types are the only distinctions that Presentation Manager makes. *Note:* Due to MS OS/2 architecture restrictions Presentation Manager may not be able to start programs running within the compatibility box. The user will be able to start them running after switching to the compatibility box and entering the command at the MS OS/2 prompt.

**Root Group** This is a predefined program group, with a reserved group handle. By default, all programs in the Installed Program List are included in this group, unless the WSHAddProgram function specifies a non-zero group handle when the program is added to the Installed Program List.

This group is normally the first list that the user will see when the Start-A-Program function is invoked. This initial view however can be changed by the user when he shuts the system down. In that case he will be presented with the same view as when the system closed.

The value of the reserved handle may change if the format of group and program handles is changed, however it will always be the value of the target data type with all bits set to zero. With the current definition of a group handle the value is 0.

The root group may contain group handles as well as program handles. (It would normally contain group handles.)

The “Name” of the root group is “WinROOT-GROUP”.

**Save-Group** This is a predefined program group, with a reserved group handle. This group will be automatically created by Presentation Manager when the “Save Configuration” option is selected from the File Cabinet. This group will be automatically started by Presentation Manager when the system initializes.

The name of this group is “WinSAVE-GROUP”.

This group is NOT for general use, and as a result has some restrictions.

- It will only contain **Program-Handles** (no groups).
- Duplicate titles will be permitted (so that the same visual appearance can be recreated).

**Selectability** A Switch List entry may be selectable or non-selectable. If a running program has its entry in the switch list marked as non-selectable then it will not be possible to reach that program using the “jump application” “hot key”.

An entry that is marked non-selectable can still be made active by directly switching to it using the WSHSwitchToProgram function, by directly selecting on a window of that application if it is a Presentation Manager application, or by bringing the Switch List to the foreground and selecting the entry for that application.

**Switch List** This is a data structure, maintained by the API, that contains information about all of the programs that are currently executing. Each executing copy of a program is given an entry in the list, The entry contains a “Program Title” for the executing program (often the same name as the Installed Program List), plus some control information - **Window Handle**, if it is a program in the Presentation Manager screen group, **Screen Group Id**, if it is in a non-Presentation Manager screen group, **Process Id** for the process within which the program is running, **Visibility** and **Selectability**.

**Switch List Handle** This is a number, assigned at the time a “program” is added to the “Switch List”, that uniquely identifies the executing copy of the program.

The number used is fixed from the time a program is added to the list until the time it is removed. The number assigned will not be used for any other program.

Most Shell API function calls that reference executing programs use the program handle to identify a program.

**Visibility** Both the Installed Program List and Switch List have the idea of a **visible** entry. When defined, this attribute determines if the appropriate entry (Switch List or Installed Program List) is displayed on the screen. It has no other effect. In particular it says nothing about the visibility of whatever windows may belong to the application.

By marking an entry as **not-visible** no entry will appear on the screen, and therefore no direct selection will be possible. Note however that it will still be possible to select the item by jumping to it using the jump application key (for Switch List entries) or by using the start program API calls for entries in the Installed Program List.

The visibility attribute is returned as part of the information provided when the entry is read using WSHQueryProgramTitles or WSHQueryDefinition.

## 3.2.2 List of functions provided by the Shell API

### 3.2.2.1 Program Use

These functions allow an application to obtain the names of all defined programs and program groups.

WSHQueryProgramHandle  
WSHQueryProgramTitles  
WSHQueryProgramType  
WSHQueryProgramUse  
WSHStartProgram

### 3.2.2.2 Adding a Program.

The following functions are used to maintain the list of programs and program groups that are available to be started.

The first group of functions maintain the information held for a single program, the smallest “executable” unit known to the Shell.

WSHQueryProgramHandle  
WSHQueryProgramType  
WSHQueryProgramTitles

The following group of functions are used to support the collection of programs into related groups. The list of programs within a group can be read using a single WSHQueryProgramTitles function.



```
WSHAddProgram  
WSHAddToGroup  
WSHCreateGroup
```

### 3.2.2.3 Switching Programs.

The following functions are used to maintain the list of currently started programs, to make one of those programs the “foreground” program, either via direct selection or through hot-key processing, and to stop an executing program.

```
WSHAddSwitchEntry  
WSHChangeSwitchEntry  
WSHQuerySwitchEntry  
WSHQuerySwitchHandle  
WinQueryTaskTitle  
WinQueryTaskSizePos  
WSHQuerySwitchList  
WSHRemoveSwitchEntry  
WSHSwitchToProgram
```

### 3.2.2.4 Clipboard

The basic Clipboard functions are provided by a window management API and do not concern the shell. The visual clipboard (CLIPBRD.EXE in MS-Windows) does not need an API and is not included in this section. A visual clipboard will be included with the Presentation Manager product.

### 3.2.2.5 Control Panel

The control panel (CONTROL.EXE) provided with MS-Windows is no more than a limited editor of the initialization file WIN.INI. The Presentation Manager control panel would be used to control similar settings for the Presentation Manager environment but would include additional function. The Control Panel API allows a program to read/write default control settings for the system and are listed below.

```
WinQueryProfileInt  
WinQueryProfileString  
WinWriteProfileString  
WSHQueryProfileSize  
WSHQueryProfileData  
WSHWriteProfileData
```

### **3.2.2.6 PrtPlot**

This section is a separate consideration.

## **3.2.3 Using the API for application programmers**

This section details how the Shell API calls are used to execute various tasks. This is not an exhaustive list, but does give some guidelines to using the Shell API calls.

### **3.2.3.1 Executing Programs**

#### *3.2.3.1.1 Starting Presentation Manager programs*

##### *Using the Start-A-Program function*

When your program has been defined in the Installed Program List, a user of the Start-A-Program function may select your program from the displayed list of programs. The program is then started in the Presentation Manager screen group as a Presentation Manager program.

When started in this way, the new program does not immediately appear in the Switch List. See the section, “Using the Switcher”, for information on creating the Switch List entry for your program.

##### *Using the DOSStartSession function*

Any application in the system can use the DOSStartSession function to cause a program in the Installed Program List to be started.

The results in this case are identical to those of selecting the program using the Start-A-Program function.

##### *Using the DOSExecPgm function*

Programs started using the DOSExecPgm function will be started in the screen group of the program issuing the function. Thus if a Presentation Manager program uses DOSExecPgm then the target program will be started in the Presentation Manager screen group and will have access to the Presentation Manager functions. A program started from another screen group will not be able to use Presentation Manager functions since it will not be running in the Presentation Manager screen group.

No switch list entry is provided for programs started in this way. See the section, “Using the Switcher”, for information on creating the Switch List entry for your program.

### *3.2.3.1.2 Starting non-Presentation Manager programs*

These are programs which must run in a separate screen group, or have not been written to the Presentation Manager API.

#### *Using the Start-A-Program function*

When your program has been defined in the Installed Program List, a user of the Start-A-Program function may select your program from the displayed list of programs. The program is started in a new screen group.

When started in this way, the new program is given a Switch List entry which is both visible and selectable. The long name of this entry is the program title as defined in the Installed Program List. See the section, "Using the Switcher", for information on changing the Switch List entry for your program.

#### *Using the DOSStartSession function*

Any application in the system can use the DOSStartSession function to cause a program defined using the Add-A-Program function to be started.

The results in this case are identical to those of selecting the program using the Start-A-Program function.

The program will be started in a new screen group.

A switch list entry is created for these programs. The entry is both visible and selectable, and will have a title equal to the Program Title supplied on the DOSStartSession call. If no Program Title is supplied then the name of the .EXE file is used, without any qualifying path information.

#### *Using the DOSExecPgm function*

Programs started using the DOSExecPgm function will be started in the screen group of the program issuing the function. Thus if a Presentation Manager program uses DOSExecPgm then the target program will be started in the Presentation Manager screen group and will have access to the Presentation Manager functions. A program started from another screen group will not be able to use Presentation Manager functions since it will not be running in the Presentation Manager screen group.

No switch list entry is provided for programs started in this way. See the section, "Using the Switcher", for information on creating the Switch List entry for your program.

### 3.2.3.2 Creating or changing the switch list entry

#### *Presentation Manager programs*

- For Presentation Manager programs started by the Start-A-Program function, by the DOSStartSession function.  
After creating a main window, it is the application program's responsibility to create a switch list entry using the WSHAddSwitchEntry function. The entry placed in the switch list can be given a default title which will be the same as the name of the program in the Installed Program List.
- For programs started by means of the DOSExecPgm function, the above options are still available, except that the default name option of the WSHAddSwitchEntry function cannot be used.

Once a switch list entry is established, it is possible to change the details of the entry at any time using the WSHChangeSwitchEntry function.

As an example, this might be done by an "Editor" program in order to have the switch list entry show the name of the file being edited.

#### *Non-Presentation Manager programs.*

- For non-Presentation Manager programs started by the Start-A-Program function, by the DOSStartSession function.  
A switch list entry is created automatically for these programs. The entry will have the title of the program as defined in the Installed Program List or Application Information File. This switch list entry will be both visible and selectable.  
Since a switch list entry is created automatically for these programs, they may not use the WSHAddSwitchEntry function, neither may WSHRemoveSwitchEntry be used. The switch list entry will be automatically deleted when the screen group terminates. WSHChangeSwitchEntry may be used.
- For non-Presentation Manager programs started by means of the DOSStartSession function, a switch list entry is created automatically. The entry will have a long name of the program title as passed in the DOSStartSession request. If no title is passed, then the name of the program .EXE file is used. This switch list entry will be both visible and selectable.
- For programs started by means of the DOSExecPgm function, no switch list entry is needed. This is because these programs are started in the same screen group as the caller, which will already have a switch list entry. A non-Presentation Manager screen group may have only one entry in the switch list.

Once a switch list entry is established, it is possible to change the details of the entry at any time using the `WSHChangeSwitchEntry` function.

As an example, this might be done by an “Editor” program in order to have the switch list entry show the name of the file being edited.

### **3.2.3.3 Installation of Presentation Manager**

When Presentation Manager is installed on top of an existing MS OS/2 system, data from existing installed programs will be extracted and copied across to the Presentation Manager format. Thus the user will not have to reinstall all his programs.

#### *3.2.3.3.1 Building the input for WSHAddProgram*

The control block structures required as input for the `WSHAddProgram` function are typically complex, they include a description of the programs execution environment and a descriptive name to aid the end user and optionally a description of the parameters required by the program to allow Presentation Manager to prompt the user for the required parameters at execution time.

To simplify the task of an application programmer who is using the `WSHAddProgram` function, a function is provided to produce a valid control block for use as input to `WSHAddProgram`. This function is `WSHQueryDefinition`.

#### *Using WSHLoadAIF*

This function will read a named AIF format file, and construct the corresponding control blocks from the information in the file. During the reading process, the files are checked for correctness, any errors in the files result in an error code and the control block structure is not created.

This control block can be passed to the `WSHAddProgram` function directly, or can be modified before use.

#### *Using WSHQueryDefinition*

This function will reconstruct the control block structure that was used to add an existing Installed Program List entry to the Initialization File.

As before, the output of this function can be used directly as input to the `WSHAddProgram` function. Notice, however, that this will result in a duplicate named entry in the Initialization File. While this is permitted, it may be confusing to the end user of the system.

### 3.2.3.4 Clipboard functions

The clipboard API is a development of the MS-Windows Clipboard API. The shell visual clipboard has no other API needs. For information on how the visual clipboard is used refer to the section, "Clipboard Viewer". For information on the Clipboard API refer to "Clipboard Functions".

### 3.2.3.5 Switching to another application

### 3.2.3.6 Using the API File Selection function

### 3.2.3.7 File System functions.

Presentation Manager sits on top of MS OS/2 kernel and uses the file system provided by DOS. All of the file system functions available under MS OS/2 kernel are still available under Presentation Manager.

## 3.2.4 Detailed Description of Shell API Functions

*Note:* In the following sections the API call definition is given in C format. To use the Shell API from assembler convert the C format call to assembler following the example below. The following two definitions are identical.

In *C* format:

```
WSHExampleCall((HANDLE) Handle, (char far *) Prognam,  
               (int) Strlength )  
HANDLE Handle      /* Program handle */  
char   ProgName[]  /* (returned) Name of program */  
int    Strlength   /* Length of program name buffer */
```

In *ASSEMBLER* format:

```
EXTRN WSHExampleCall:FAR  
  
PUSH WORD    Handle      ; Example handle  
PUSH@ ASCIIZ ProgName    ; (returned) ASCIIZ string  
PUSH WORD    Strlength   ; Maximum length of string  
CALL WSHExampleCall
```

On return **AX** contains the return code (non-0 for an ERROR)

### 3.2.5 Program Use API

```
short WSHQueryProgramHandle( (char far *)ProgName,  
                             (HANDLE far *)&ProgHandle, (short)ProgCount)  
char      ProgName[] /* Program names */  
HANDLE far * ProgHandle[] /* Array of program handles */  
short      ProgCount /* Maximum count of handles */
```

---

**Purpose** Given the name of an executable file, this function will return the program handles using that executable file. This will allow the system to determine where a particular executable file is being used.

**Where:** **ProgName** is the name of the executable file. If it is a fully qualified name it must match exactly with the stored name. If it is an unqualified executable name (no path) then a match will occur on every instance of a executable file of that name.

**ProgHandle** is an array of handles. The user controls how many handles he wants to see by the **ProgCount** parameter.

**ProgCount** is the dimension of the array, and thus the maximum number of handles that can be returned to the user.

**Returns:**

```
Count = 0    No Handles matched OR Error  
           (use WinGetLastError)  
Count <> 0    The count of handles in the array
```

- No program found with given name
- Invalid path statement

**Remarks**

```
long WSHQueryProgramType( (char far *)ProgramName)  
char  ProgramName[] /* Name of program being queried */
```

---

**Purpose.**

Get the program type for a specified executable file. The valid types are:

- Presentation Manager
- Vio-windowed
- Real

- Non-Presentation Manager
- Program\_Group
- Unknown

Where: **ProgramName** is the qualified or unqualified name of a DOS executable file.

Returns:

```
ProgramType = 0    No name matched OR Error
             <> 0    The first matching program handle
```

Remarks

If more than one program exists that matches the input program name, then the type of the first program found will be returned. It is expected that programs will always be used in the same environment (e.g. Presentation Manager, OR non-Presentation Manager but not both).

```
int WSHQueryProgramTitles( (HANDLE)GroupHandle, (GISPTR)Buffer,
                           (int)BufferLen)
HANDLE   GroupHandle /* Handle of group to read */
GISSTRUCT Buffer      /* (returned) Data Buffer */
int      BufferLen    /* Length of buffer in bytes */
```

---

Purpose Used to obtain information about programs and program groups defined as “startable” to the Shell. Information about all programs in a group is returned in a single operation. The information returned is an array of entries, one for each program in the group.

Where: **GroupHandle** is the handle of the group for which information is required, as returned in a previous WSHQueryProgramTitles request. The group handle may be one of the reserved group handles. If GroupHandle is WinROOT-GROUP, then information for the root group is returned. If GroupHandle is WinMASTER-GROUP, then information for all installed programs is returned up to the size of the buffer provided.

**Buffer** is an area of storage into which Group Information Structure data structure will be placed. The format of the Group Information Structure is given below.

**BufferLen** is the maximum usable space in the buffer.

#### Definition

```
typedef struct gis {
    short  TotalNumber;
    short  ArrayCount;
    struct ProgramEntry ProgramArray[ArrayCount];
} GISSTRUCT *GISPTR;
```



*Where*

**TotalNumber** is the total count of entries in the selected group.

**ArrayCount** is the number of entries for which there was room in the buffer.

**ProgramArray** is an array of structures, one array element for each program entry within the group. The format of the structure is given below.

#### Definition

```
typedef struct ProgramEntry {
    HANDLE    ProgramHandle;
    PROGTYP   ProgramType;
    /* Program/Group Handle Flag */
    char      InvisibleFlag;
    /* zero if ENTRY is visible on screen,
       non-zero if hidden */
    char      IconFileName[&maxpath1.+1];
    char      ProgramTitle[60+1];
} PROGRAMENTRY *PROGRAMENTRYPTR;
```

*Where*

**ProgramHandle** is the program/group handle.

**ProgramType** is **PROGRAM** or **PROGRAMGROUP**.

In the case of **PROGRAM** then the information as to what type of program this is is included. The type can be Presentation Manager, non-Presentation Manager-Windowed, and non-Presentation Manager-other.

**InvisibleFlag** is **VISIBLE** if this entry appears when this group is displayed by the Shell, or **INVISIBLE** if the entry is not shown in the visible list. It does NOT refer to the visibility status of any windows belonging to this entry.

**IconFileName** is a far pointer to an ASCIIZ filename string which is where the icon definition for this entry can be found. The pointer may be NULL in which case no icon is defined.

**Program Title** is a character array containing the program title for this entry. The maximum length is 60. No leading or trailing blanks are preserved by the Shell API.

Returns:

IF ERROR ( AX not = 0 )

AX = Error Code

- Invalid handle.

- Insufficient space to contain Group Information Structure
- Invalid Group Information Structure buffer size

**Remarks**

This function can be used to find out the number of entries within a group by passing a buffer of length 2 bytes. The function will return the total number of entries within the group.

Values of BufferLen less than 2 are invalid.

If the buffer is not large enough to contain the Group Information Structure an error will be reported and as many complete array entries as possible will be placed into the buffer.

The handle specified can also be a program handle, in which case the buffer will only contain the entry for one program. Thus this call may be used to get the program title.

The list of returned program entries may contain group handles. This allows the tree structure to be built up by the caller. Note though, that information from only ONE level of the tree structure is returned by this call.

### 3.2.6 Adding a Program API

```
HANDLE WSHAddProgram( (PIBSTRUCT far *)ProgramInfo,  
                      (HANDLE)GroupHandle)  
PIBSTRUCT ProgramInfo /* Program Information Block */  
HANDLE      GroupHandle /* Target group handle */
```

---

**Purpose** This function is used to create a new program entry in the Installed Program List, and to provide the initial information about the program.

**Where:** **ProgramInfo** is a structure containing the information required by Presentation Manager to control the starting and switching to this program. The layout of the structure is given below. Specifically, the title for the program is contained within the Program Information Block.

**GroupHandle** is the handle, returned by WSHCreateGroup, of the group to which this program is to be added.

**ProgHandle** is a word in which the newly generated handle for this program will be returned.

**Definition**

```
typedef struct xywinsize {  
    short    XPos; /* X position of Window */
```

```
short    YPos;        /* Y position of Window */
short    XSize;       /* X extent of Window */
short    XSize;       /* Y extent of Window */
word     WinFlag;     /* MINIMISED or MAXIMISED
                      or INVISIBLE or NORMAL (0) */
} XYWINSIZE *XYWINSIZEPTR;
```

### Definition

```
typedef struct pib {
    PROGTYPED ProgramType;
                /* Presentation Manager, non-Presentation Manager,
                non-Presentation Manager-other,
                Group, and Visibility attribute */
    char       ProgramTitle[60+1];
    char       IconFileName[&maxpathl.+1];
    char       ExecutableName[&maxpathl.+1];
    char       StartupDirectory[&maxpathl.+1];
    XYWINSIZE  InitialPosition;
    char       HelpString[&helpstrl.+1];
    short      EnvironLength;
    char       EnvironString[EnvironLength];
    short      ParameterLength;
    char       ParameterString[1];
} PIBSTRUCT *PIBSTRUCTPTR;
```

### Where

**ProgramType** defines the type and visibility of this program. If this is a PROGRAM then the type can also be Presentation Manager, non-Presentation Manager-Windowed, and non-Presentation Manager-other. If this is a PROGRAMGROUP, then the type of program has no meaning. The Visibility attribute defines whether this entry is visible in the Start-A-Program list.

**ProgramTitle** is the title for this program. No leading or trailing blanks are preserved by the Shell API.

**IconFileName** defines the icon file associated with this program.

**ExecutableFileName** defines the executable file that will be run when this program is started.

**StartupDirectory** defines the subdirectory that will be the current drive and directory when the program starts running.

**InitialPosition** is the suggested position and size to be used on the first WINCreateWindow call. If all values are 0, then the Shell will provide an initial size and position.

**HelpString** is a short informative piece of help information for this program. This text will be displayed whenever general help is requested for this program.

**EnvironLength** is the length of the **EnvironString**.

**EnvironString** defines the environment variables to be passed to the program when it is started. This string is in the format required by DOS, i.e. a set of ASCIIZ strings, the complete set of which is terminated by a null character.

**ParameterLength** is the length of the **ParameterString**.

**ParameterString** defines the parameter to be passed to the program, via its “Command Line” when it is invoked.

Each entry in **ParameterArray** is a structure, as follows.

#### Definition

```
typedef struct parameterdescriptor {  
    WORD    ParameterType;  
    BYTE    ParameterSubtype;  
    short   PstringLength;  
    char     ParameterString[PstringLength];  
            /* Initial path for FileSystem  
             * when User Selected File Name  
             * OR default input string */  
} PARAMETERDESCRIPTOR *PARAMETERDESCRIPTORPTR;
```

**ParameterType** is one of three basic types.

- File name

When **ParameterSubtype** will be one of the following

- User selected - by definition, must exist
- User entered - must exist
- User entered - must not exist
- User entered - may or may not exist

- Free format
- Constant

**ParameterSubtype** is one of the defined filename subtypes if **ParameterType** is File-Name, otherwise it is undefined.

**PStringlength** is the length of **ParameterString** excluding the terminating zero. This is the length before any substitutions have taken place.

**ParameterString** is an ASCIIZ input string for the program. The meaning of **ParameterString** depends on **ParameterType**. These meanings are shown below.

---

Type	Meaning of ParameterString
constant	Input Parameter for the program.

user selected filename

Initial path to use for file selection when calling the FileSystem.

free format

String from which the input parameter string is constructed, which may include prompting the user for the actual parameter.

Returns:

IF ERROR ( AX not = 0 )

AX = Error Code

- Invalid Program Information Block
- Insufficient space to add Installed Program List entry
- Invalid group handle
- Title already in use in target group

Remarks

Program titles need not be unique, each call to this function will add a new definition with the same title. Note though that duplicate titles are NOT allowed within a group. (The same title can be used, but the entries must be in different groups.) *Although the same title is permitted, it may lead to confusion, and is not recommended practice.* The restriction that duplicate titles are not allowed within a group does ensure that if the same name appears in the listing of a group it is because the program has been added multiple times. (So that when the group is executed, the program in question is started multiple times.)

The value passed in **GroupHandle** must be the handle of a defined group. The “Root” group is predefined with a handle of 0. The program may not be added explicitly to the “Master-List”.

The initial window position and size information will be made available to the application by an API call. (Win-QueryTaskSizePos) The application may ignore the suggested defaults if it so wishes, otherwise it may use the defaults on the first WINCreateWindow call.

The initial position and size fields should be 0 for a non-Presentation Manager application. If no defaults are required, then both initial X and Y extents must be set to 0. In this case Start-A-Program will generate a default starting size and position whenever the application is started.

It is recommended that the initial size and position be specified as all 0s, whereupon the Shell will position the window with regard to other programs. The values in this

structure are device dependent and should ONLY be specified if the target device is known and cannot change.

The visibility attribute is returned as part of the information provided in the WSHQueryProgramTitles function. An “invisible” group or program will not appear in the Start-A-Program list and so cannot be started by user selection.

```
int WSHChangeProgram( (HANDLE)ProgHandle,
                      (PIBSTRUCT far *)ProgramInfo)
HANDLE   ProgHandle   /* Program to be changed */
PIBSTRUCT ProgramInfo /* Changed Program Information Block */
```

**Purpose** To replace an existing program entry in the Installed Program List with a complete new definition.

**Where:** **ProgHandle** is the handle returned by WSHAddProgram when the program was first added to the system.

**ProgramInfo** is a PIBSTRUCT data structure which is fully defined in the WSHAddProgram function. The format is:

#### Definition

```
typedef struct pib {
    PROGTYPE ProgramType;
                /* Presentation Manager, non-Presentation Manager
                non-Presentation Manager-other,
                Group, and Visibility attribute */
    char      ProgramTitle[60+1];
    char      IconFileName[&maxpath1.+1];
    char      ExecutableName[&maxpath1.+1];
    char      StartupDirectory[&maxpath1.+1];
    XYWINSIZE InitialPosition;
    char      HelpString[&helpstr1.+1];
    short     EnvironLength;
    char      EnvironString[EnvironLength];
    short     ParameterLength;
    char      ParameterString[1];
} PIBSTRUCT *PIBSTRUCTPTR;
```

#### Where

**ProgramType** defines the type and visibility of this program. If this is a PROGRAM then the type can also be Presentation Manager, non-Presentation Manager-Windowed, and non-Presentation Manager-other. If this is a PROGRAMGROUP, then the type of program has no meaning. The Visibility attribute defines whether this entry is visible in the Start-A-Program list.

**ProgramTitle** is the title for this program. No leading or trailing blanks are preserved by the Shell API.

**IconFileName** defines the icon file associated with this program.

**ExecutableFileName** defines the executable file that will be run when this program is started.

**StartupDirectory** defines the subdirectory that will be the current drive and directory when the program starts running.

**InitialPosition** is the suggested position and size to be used on the first WINCreateWindow call. If all values are 0, then the Shell will provide an initial size and position.

**HelpString** is a short informative piece of help information for this program. This text will be displayed whenever general help is requested for this program.

**EnvironLength** is the length of the **EnvironString**.

**EnvironString** defines the environment variables to be passed to the program when it is started. This string is in the format required by DOS, i.e. a set of ASCII strings, the complete set of which is terminated by a null character.

**ParameterLength** is the length of the **ParameterString**.

**ParameterString** defines the parameter to be passed to the program, via its “Command Line” when it is invoked.

Each entry in **ParameterArray** is a structure, as follows.

#### Definition

```
typedef struct parameterdescriptor {
    WORD      ParameterType;
    BYTE      ParameterSubtype;
    short      PstringLength;
    char       ParameterString[PstringLength];
               /* Initial path for FileSystem
                  when User Selected File Name
                  OR default input string */
} PARAMETERDESCRIPTOR *PARAMETERDESCRIPTORPTR;
```

**ParameterType** is one of three basic types.

- File name

When **ParameterSubtype** will be one of the following

- User selected - by definition, must exist
- User entered - must exist
- User entered - must not exist
- User entered - may or may not exist

- Free format
- Constant

**ParameterSubtype** is one of the defined filename subtypes if **ParameterType** is File-Name, otherwise it is undefined.

**PStringlength** is the length of **ParameterString** excluding the terminating zero. This is the length before any substitutions have taken place.

**ParameterString** is an ASCIIZ input string for the program. The meaning of **ParameterString** depends on **ParameterType**. These meanings are shown below.

---

Type	Meaning of ParameterString
constant	Input Parameter for the program.
user selected filename	Initial path to use for file selection when calling the FileSystem.
free format	String from which the input parameter string is constructed, which may include prompting the user for the actual parameter.

Returns:

```
IF ERROR ( AX not = 0 )  
    AX = Error Code
```

- Invalid program or program group handle
- New title already exists in group
- Invalid Program Information Block

Remarks

This function performs a complete replacement of the information stored for a program; no information is carried over from the existing definition.

This function may be used to change the title or visibility of a program group. In this case the **PROGTYPE** field in the Program Information Block will indicate that the handle is a program group handle, and the Program Information Block will only contain the **ProgramType** and **ProgramTitle** fields.

The group associations for the program being changed are not affected by this function.

**WSHQueryDefinition** can be used to obtain a working copy of the current definition. This copy can be changed and used as input to this function.



There is no facility provided for writing the changed information out to a new Application Information File (AIF).

```
int WSHQueryDefinition( (HANDLE)ProgHandle,
                        (PIBSTRUCTPTR)Buffer, (int)BufferLen)
HANDLE   ProgHandle      /* Handle of program selected */
PIBSTRUCT Buffer          /* Buffer for Program Information Block */
int      BufferLen        /* Length of Buffer */
```

**Purpose** To obtain a copy of the Program Information Block held for a program, or program group. The program for which information is required is identified by its handle. This function can be used to read the definition for a program group. Note however that a group definition **ONLY** consists of the **PROGTYPE** and **ProgramTitle** fields.

**Where:** **ProgHandle** is the handle returned when the program was added to the list of programs using the **WSHAddProgram** function.

**Buffer** is an area of storage into which the routine will construct the Program Information Block for this program.

**BufferLen** is the maximum usable space in the buffer that is to be used for the Program Information Block.

The format of a Program Information Block is shown below. For descriptions of the field contents within a Program Information Block.

### Definition

```
typedef struct pib {
    PROGTYPE  ProgramType;
              /* Presentation Manager, non-Presentation M
              non-Presentation Manager-other,
              Group, and Visibility attribute */
    char      ProgramTitle[60+1];
    char      IconFileName[&maxpath1.+1];
    char      ExecutableName[&maxpath1.+1];
    char      StartupDirectory[&maxpath1.+1];
    XYWINSIZE InitialPosition;
    char      HelpString[&helpstr1.+1];
    short     EnvironLength;
    char      EnvironString[EnvironLength];
    short     ParameterLength;
    char      ParameterString[1];
} PIBSTRUCT *PIBSTRUCTPTR;
```

*Where*

**ProgramType** defines the type and visibility of this program. If this is a **PROGRAM** then the type can also be **Presentation Manager**, **non-Presentation Manager-Windowed**, and **non-Presentation Manager-other**. If this is a

PROGRAMGROUP, then the type of program has no meaning. The Visibility attribute defines whether this entry is visible in the Start-A-Program list.

**ProgramTitle** is the title for this program. No leading or trailing blanks are preserved by the Shell API.

**IconFileName** defines the icon file associated with this program.

**ExecutableFileName** defines the executable file that will be run when this program is started.

**StartupDirectory** defines the subdirectory that will be the current drive and directory when the program starts running.

**InitialPosition** is the suggested position and size to be used on the first WINCreateWindow call. If all values are 0, then the Shell will provide an initial size and position.

**HelpString** is a short informative piece of help information for this program. This text will be displayed whenever general help is requested for this program.

**EnvironLength** is the length of the **EnvironString**.

**EnvironString** defines the environment variables to be passed to the program when it is started. This string is in the format required by DOS, i.e. a set of ASCIIZ strings, the complete set of which is terminated by a null character.

**ParameterLength** is the length of the **ParameterString**.

**ParameterString** defines the parameter to be passed to the program, via its "Command Line" when it is invoked.

Each entry in ParameterArray is a structure, as follows.

#### Definition

```
typedef struct parameterdescriptor {
    WORD    ParameterType;
    BYTE    ParameterSubtype;
    short   PstringLength;
    char    ParameterString[PStringLength];
           /* Initial path for FileSystem
            * when User Selected File Name
            * OR default input string */
} PARAMETERDESCRIPTOR *PARAMETERDESCRIPTORPTR;
```

**ParameterType** is one of three basic types.

- File name

When ParameterSubtype will be one of the following

- User selected - by definition, must exist
- User entered - must exist
- User entered - must not exist
- User entered - may or may not exist

- Free format
- Constant

**ParameterSubtype** is one of the defined filename subtypes if **ParameterType** is File-Name, otherwise it is undefined.

**PStringlength** is the length of **ParameterString** excluding the terminating zero. This is the length before any substitutions have taken place.

**ParameterString** is an ASCIIZ input string for the program. The meaning of **ParameterString** depends on **ParameterType**. These meanings are shown below.

---

Type	Meaning of ParameterString
constant	Input Parameter for the program.
user selected filename	Initial path to use for file selection when calling the FileSystem.
free format	String from which the input parameter string is constructed, which may include prompting the user for the actual parameter.

#### Returns:

IF ERROR ( AX not = 0 )

AX = Error Code

- Invalid Program Handle
- Invalid Program Information Block buffer size
- Insufficient space to contain Program Information Block

#### Remarks

If the buffer is not large enough to contain the Program Information Block an error will be reported, and the space that would have been required for the Program Information Block (in bytes) is placed in the first word of the buffer provided.

If the buffer is large enough the first word will be set to the total number of bytes used.

The minimum possible size for this buffer is 2 bytes, in which case only the numeric result field will be provided, thus indicating the actual space required for successful completion.

If the handle is a program handle, as opposed to a group handle, the resulting information is in a format suitable for input to the WSHAddProgram and WSHChangeProgram functions.

If the handle is a program group handle, then the structure only contains the PROGTYPE and ProgramTitle fields.

```
int WSHRemoveProgram( (HANDLE)ProgHandle)
HANDLE ProgHandle    /* Handle of Program to Remove */
```

---

**Purpose** To erase the definition of a program from the Installed Program List. All references to the program being removed, from all groups, are also removed.

**Where:** **ProgHandle** is the handle returned for the program when it was added to the list by the WSHAddProgram function.

**Returns:**

IF ERROR ( AX not = 0 )

AX = Error Code

- Invalid Program Handle

**Remarks**

You cannot use this routine to remove a group definition. To remove a group definition use WSHDestroyGroup.

```
int WSHAddToGroup( (HANDLE)GroupHandle, (HANDLE)ProgHandle)
HANDLE GroupHandle    /* Group handle */
HANDLE ProgHandle     /* Program Handle */
```

---

**Purpose** To add the definition of a program or program group to an existing program group.

**Where:** **GroupHandle** is a word which contains the group handle to which the program will be added.

**ProgHandle** is the handle of the program or program group to be added.

**Returns:**

IF ERROR ( AX not = 0 )

AX = Error Code

- Invalid target group handle
- Invalid source program or program group handle
- Insufficient space to add Installed Program List entry
- Circular reference not allowed
- Duplicate title

**Remarks**

The target group must already exist.

```
HANDLE WSHCreateGroup( (char far *)GroupTitle,
                       (char)GroupVis, (HANDLE)TargetGroupH,
                       (char far *)GroupHelp)
char    GroupTitle[]    /* Name of new group */
char    GroupVis        /* Visibility Option */
HANDLE  TargetGroupH    /* Target group handle */
char    GroupHelp[]     /* Help for group */
```

---

**Purpose** To add the definition of a new group of programs to the program list. When the group is created, it will not contain any program references, the WSHAddToGroup function must be used to place program references into a group.

**Where:** **GroupTitle** is an ASCIIZ string which contains the readable text name of the group being defined. If the title is longer than 60 characters, then the title will be truncated.

The title must contain valid ASCII characters; note that "\" is not permitted, and the string must be non-null (at least one non-blank character). Leading and trailing blanks are removed by the API before further validating the title.

**GroupVis** controls the visibility attribute for the group entry.

**TargetGroupH** is the group into which this group is to be placed. In order to have the group appear in the Root group specify a handle of 0.

**GroupHelp** is a short piece of help information for this program group. It is optional (i.e. the pointer to the string may be null) but if specified the string must be non-null, with at least one non blank character.

**GroupHandle** is a HANDLE in which the newly generated handle for this group will be returned.

**Returns:**

```
IF ERROR ( AX not = 0 )
    AX = Error Code
```

- Invalid title
- Invalid target group handle
- Duplicate title
- Insufficient space to add Installed Program List entry

**Remarks**

The visibility attribute is returned as part of the information provided in the WSHQueryProgramTitles function. An “invisible” group or program will not appear in the Start-A-Program list and so cannot be started by user selection.

The Shell Start-A-Program program will honor the visibility attribute.

If the group already exists, then the existing group handle will be returned. No new handle will be created.

```
int WSHDestroyGroup( (HANDLE)GroupHandle)
HANDLE GroupHandle  /* Handle of group to remove */
```

---

**Purpose** To remove a group definition from the Installed Program List, and also to remove the associations between the removed group and any program entries in the group.

**Where:** **GroupHandle** is the handle of the group to be removed.

**Returns:**

```
IF ERROR ( AX not = 0 )
    AX = Error Code
```

- Invalid group handle.

**Remarks**

It is not permitted to remove either the “Root” group or “Master-List”.

```
int WSHRemoveFromGroup( (HANDLE)GroupHandle, (HANDLE)ProgHandle)
HANDLE GroupHandle     /* Group handle */
HANDLE ProgHandle      /* Program Handle */
```

---

**Purpose** To remove the definition of a program from a group in the Installed Program List.

**Where:** **GroupHandle** is the handle of the group from which the program association is to be removed.

**ProgHandle** is the handle of the program for which the association is to be removed.

**Returns:**

```
IF ERROR ( AX not = 0 )
```

```
    AX = Error Code
```

- Invalid group handle
- Handle does not exist within group

**Remarks**

It is not possible to remove the association between the “Master-List” and a program.

If the same program appears multiple times within a group (which must have been caused by multiple WSHAddToGroup calls) then WSHRemoveFromGroup will only remove one definition. WSHRemoveFromGroup will have to be called as many times as WSHAddToGroup was called to delete the program entirely from the group.

### 3.2.7 Switching Programs API

```
HANDLE WSHAddSwitchEntry((SWCNTRL far *)Control)
SWCNTRL far * Control    /* Switch list control information */
```

---

**Purpose** Add an entry to the switch list. The entry need not necessarily be made visible, but it must refer to a process that is currently running.

**Where:** **Control** is a structure with the following format:

**Definition**

```
typedef struct swcontrol {
    char    SwTitle[60+1];
    HANDLE  WindowHandle;    /* 0 if no window */
    HANDLE  IconHandle;      /* 0 if no icon */
    HANDLE  ProgramHandle    /* 0 if not an installed prog */
    WORD    ProcessId;       /* For this entry */
    WORD    SessionId;       /* For this entry */
    BYTE    SwitchFlagA;
    BYTE    SwitchFlagB;
} SWCNTRL *SWCNTRLPTR;
```

*Where*

**SwTitle** is the ASCIIZ title for this entry. It must be a non-null string in order to define a valid switch list entry.

**WindowHandle** is the handle of the main window belonging to this entry. (0 means that there is no window, in which case this is a non-Presentation Manager entry.)

**IconHandle** is the handle to access the icon for this entry.

**ProgramHandle** is the program handle that was used to start the program resulting in this entry in the switch list. Only items in the switch list that have a program handle entry can be included in the *SAVE CONFIGURATION* function as these are the only programs that the Shell knows how to get started.

**ProcessId** is the DOS process id for this entry.

**SessionId** is the session identifier for this entry. The session id is generated by DOS when starting a new session running. This is a read only field and may NOT be changed by the user.

**SwitchFlagA** defines the visibility of an entry in the Switch List. It is a byte with **ONE** of the following values:

<b>NOT_VISIBLE</b>	No entry in the visual list
<b>NOT_SELECTABLE</b>	Visible, but disabled (grayed)
<b>SWL_NORMAL</b>	Visible and enabled (default value == 0)

- **NOT\_VISIBLE** means that this entry will not appear in the visual switch list, but can be selected by the user by using the **Jump-Application** hotkey, or, if it is a Presentation Manager application, making a direct selection on one of the windows belonging to that application.
- **NOT\_SELECTABLE** means that this entry is not enabled for selection. (On a colour screen it will appear grayed.)
- **SWL\_NORMAL** means that this entry is both visible and enabled for selection. (This is the default when 0 is given.)

**SwitchFlagB** defines the response of this entry to the **Jump-Application** hotkey. It is a byte with **ONE** of the following values:

<b>JUMP_ENABLED</b>	Will appear in the round robin sequence
<b>JUMP_DISABLED</b>	Will not appear in the round robin sequence

- **JUMP\_ENABLED** means that this entry will be activated as a result of pressing the **Jump-Application** hotkey.
- **JUMP\_DISABLED** means that this entry will not be activated as a result of pressing the **Jump-Application** hotkey. The application can still be activated by directly selecting the entry in the Switch List (if the entry is visible and enabled) or, if it is a Presentation Manager application, by making a direct selection on one of the windows belonging to that application.

**SwHandle** is the handle to the switch list entry for this program and should be used on subsequent switch list



processing calls.

Returns:

IF ERROR ( AX not = 0 )

AX = 0 implies error (use WinGetLastError call)  
AX/DX = SwHandle

- (Presentation Manager) Main Window already in switch list
- (Non-Presentation Manager) Screen group already in switch list
- Invalid window handle
- Not a main window handle
- Invalid parameter block
- Invalid icon handle
- Invalid name
- Switch list full
- Invalid Process Id

Remarks

If a null string is passed for the entry name Presentation Manager will use the name under which the application was started. This ONLY applies to Presentation Manager applications, and only to the first WSHAddSwitchEntry for that application. Otherwise a null name (i.e. a name consisting of only the terminating zero) is invalid.

Leading and trailing blanks will be removed from the program title. This ensures a consistent user view. If the program title is longer than 60 bytes it will be truncated.

There is an implementation maximum of the number of Switch List entries. However the maximum number will be several hundred programs and there should be no reason to reach the limit. Before the limit is reached other machine limitations will have been encountered.

The program handle will be passed across to the Switch List and will mean that the program started in this manner can be stored during shutdown of the machine and restored later by the Shell restore configuration function.

If a program has been registered with the WSHSwitchProgramRegister call then this API will send a message to the window defined by the WSHSwitchProgramRegister. This message has the following format:

SH\_SWITCHLIST

lParam1:        DWORD idAddEntry  
lParam2:        HANDLE Switch Handle  
Returns:        OL

---

#### Description

This message is used by all the Task Manager routines to inform the visual shell that a new entry has been made to the Switch List.

This message is always queued.

The handle is the new switch list handle.

```
int WSHChangeSwitchEntry((HANDLE)SwHandle, (SWCNTRL far *)SwControl)
HANDLE    SwHandle        /* Switch List Handle */
SWCNTRL far * SwControl   /* Switch list control block */
```

---

**Purpose**    Alter the attributes and/or name of an entry in the switch list.

**Where:**   **SwHandle** is the switch list entry handle whose information is to be changed.

**SwControl** is the same structure defined for WSHAddSwitchEntry and has the following form.

#### Definition

```
typedef struct swcontrol {
    char    SwTitle[60+1];
    HANDLE WindowHandle;        /* 0 if no window */
    HANDLE IconHandle;         /* 0 if no icon */
    HANDLE ProgramHandle       /* 0 if not an installed program */
    WORD    ProcessId;         /* For this entry */
    WORD    SessionId;         /* For this entry */
    BYTE    SwitchFlagA;
    BYTE    SwitchFlagB;
} SWCNTRL *SWCNTRLPTR;
```

#### Where

**SwTitle** is the ASCIIZ title for this entry. It must be a non-null string in order to define a valid switch list entry.

**WindowHandle** is the handle of the main window belonging to this entry. (0 means that there is no window, in which case this is a non-Presentation Manager entry.)

**IconHandle** is the handle to access the icon for this entry.

**ProgramHandle** is the program handle that was used to start the program resulting in this entry in the switch list. Only items in the switch list that have a program handle entry can be included in the *SAVE CONFIGURATION* function as these are the only programs that the Shell knows how

to get started.

**ProcessId** is the DOS process id for this entry.

**SessionId** is the session identifier for this entry. The session id is generated by DOS when starting a new session running. This is a read only field and may NOT be changed by the user.

**SwitchFlagA** defines the visibility of an entry in the Switch List. It is a byte with **ONE** of the following values:

<b>NOT_VISIBLE</b>	No entry in the visual list
<b>NOT_SELECTABLE</b>	Visible, but disabled (grayed)
<b>SWL_NORMAL</b>	Visible and enabled (default value == 0)

- **NOT\_VISIBLE** means that this entry will not appear in the visual switch list, but can be selected by the user by using the **Jump-Application** hotkey, or, if it is a Presentation Manager application, making a direct selection on one of the windows belonging to that application.
- **NOT\_SELECTABLE** means that this entry is not enabled for selection. (On a colour screen it will appear **grayed**.)
- **SWL\_NORMAL** means that this entry is both visible and enabled for selection. (This is the default when 0 is given.)

**SwitchFlagB** defines the response of this entry to the **Jump-Application** hotkey. It is a byte with **ONE** of the following values:

<b>JUMP_ENABLED</b>	Will appear in the round robin sequence
<b>JUMP_DISABLED</b>	Will not appear in the round robin sequence

- **JUMP\_ENABLED** means that this entry will be activated as a result of pressing the **Jump-Application** hotkey.
- **JUMP\_DISABLED** means that this entry will not be activated as a result of pressing the **Jump-Application** hotkey. The application can still be activated by directly selecting the entry in the Switch List (if the entry is visible and enabled) or, if it is a Presentation Manager application, by making a direct selection on one of the windows belonging to that application.

Returns:

```
IF ERROR ( AX not = 0 )
    AX = Error Code
```

- Invalid Handle
- Invalid Icon Handle
- Invalid Program Title (it cannot be zero length)
- Invalid SWCNTRL block
- Invalid Process Id

**Remarks**

The maximum length of a program title in the switch list is 60. Any entry longer than this will have the title truncated with no warning.

If a program has been registered with the WSHSwitchProgramRegister call, then this API will send a message to the window defined by the WSHSwitchProgramRegister. This message has the following format:

```
SH_SWITCHLIST
lParam1:      DWORD idChangeEntry
lParam2:      HANDLE Switch Handle
Returns:      OL
```

---

**Description**

This message is used by all the Task Manager routines to inform the visual shell that an entry in the Switch List, has been changed.

This message is always queued.

The handle is the changed switch list handle.

```
int WSHQuerySwitchEntry( (HANDLE) SwHandle, (SWCNTRLPTR) Buffer)
HANDLE SwHandle /* Switch List Handle */
SWCNTRL far * Buffer /* Buffer for definition */
```

---

**Purpose** To obtain a copy of the Switch List Control block for a specific application. The application for which information is required is identified by its handle.

**Where:** **SwHandle** is the handle returned when the program was added to the switch list using the WSHAddSwitchEntry function.

**Buffer** is an area of storage into which the routine will construct the switch list entry for this program.

**Returns:**

IF ERROR ( AX not = 0 )

AX = Error Code

- No entry corresponding to handle

Remarks

If **SwHandle** cannot be found, for whatever reason, then an error is returned.

This call is available to non-Presentation Manager applications.

```
HANDLE WSHQuerySwitchHandle( (HANDLE)WHandle, (WORD)ProcessID)
HANDLE  WHandle              /* Window Handle */
WORD    ProcessID            /* Process ID */
```

---

**Purpose** Find the switch list handle belonging to a specific process or window. This is used by those applications that know what the window handle is (via some other means) and want to alter or query the attributes of the entry for this window in the switch list.

**Where:** **WHandle** is a window handle for an application running in the Presentation Manager screen group for which the switch list handle is required *or* 0.

**ProcessID** is the process id for which a switch list handle is required *or* 0. This is mutually exclusive with the first parameter and is designed for use by non-Presentation Manager applications which do not have a window handle.

**SwHandle** is set to the switch list handle for this entry, if the entry exists (i.e. AX  $\neq$  0). If no matching entry can be found in the switch list then 0 is returned.

Returns:

IF ERROR ( AX = 0 )

Use WinGetLastError

- Invalid handle
- Invalid process id
- Invalid parameters
- No entry in switch list

Remarks

This entry could be used by a parent application after starting a child in order to change or modify the switch list entry for the child, or even to use the shell facilities to kill the child. For example, it may well be the easiest way to prompt the user to confirm the death of a child process.

If both window handle and process id are provided they must

be consistent with each other. Otherwise NO handle will be returned.

```
int WinQueryTaskTitle( (word)ProcessID, (char far *)NameBuffer,  
                      (int)BufferLen)  
word    ProcessID      /* Process ID */  
char far * NameBuffer  /* (returned) Program name */  
int     BufferLen       /* Maximum buffer length */
```

---

**Purpose** Find the title which this program has in the Switch List. This is used by those applications that want to know what string the user associates with this program. The application can then use the same string in its window and/or switch list entry. The main purpose of the call is before an application creates its first window, when it can insert the same title into the window and Switch List as appears in the Start-A-Program list.

**Where:** **ProcessID** is the process id to identify the application whose title is required.

**NameBuffer** is filled with the name of this process, if the program is known by the shell.

**BufferLen** is the maximum length of NameBuffer. If the name is too long to fit into the buffer the name will be truncated. The maximum number of characters of title that can be copied to the buffer is **BufferLen-1** as there must always be room for a terminating zero.

**Returns:**

IF ERROR ( AX not = 0 )

AX = Error Code

- process id is not known
- Title truncated

**Remarks**

Use this entry to get the title associated with a program.

If found, the name will be copied to the supplied buffer, with a terminating zero added.

```
int WinQueryTaskSizePos( (word)ProcessID, (XYWINSIZE far *)PositionBlock)  
word    ProcessID      /* Process ID */  
XYWINSIZE PositionBlock /* (returned) Size and Position */
```

---

**Purpose** Find the size and position for the initial window.

Where: **ProcessID** is the process id to identify the application whose title is required.

**PositionBlock** is filled with the size and position to use when creating the window. PositionBlock has the following structure:

#### Definition

```
typedef struct xywinsize {
    short  XPos;      /* X position of Window */
    short  YPos;      /* Y position of Window */
    short  XSize;     /* X extent of Window */
    short  YSize;     /* Y extent of Window */
    word   WinFlag;   /* MINIMISED or MAXIMISED
                      or INVISIBLE or NORMAL (0) */
} XYWINSIZE *XYWINSIZEPTR;
```

Returns:

IF ERROR ( AX not = 0 )

AX = Error Code

- process id is not known

Remarks

Use this entry to get the suggested size and position to use for the first window created. The system will provide default values if there are no values stored for this particular program. (i.e. there is no program handle, because a .EXE or .CMD file was started.)

If a window size and position needs to be generated by the shell (i.e. Switch List) the units returned will be those that can be used on the Presentation Manager create window API calls.

The values returned will be for the **frame** window, not the client area. The size will be large enough to provide a non-null client area which will be no more than one quarter of the available windowing area. The exact algorithm to be used will not form part of the specification.

```
short WSHQuerySwitchList((SWBlock far *)Buffer, (int)Length)
SWBlock far * Buffer /* Buffer into which to copy data */
int Length /* Length of buffer in bytes */
```

---

**Purpose** To collect information about entries defined in the switch list. Information about all entries in the switch list is returned in a single operation, each individual entry being defined by a single array element.

Where: **Buffer** will be set to the Switch List Block as described below.

**Length** is the length in bytes of the buffer. If Length=0 then only the number of entries in the list is returned. This is also the case if a null address is given for **Buffer**.

### Definition

```
typedef struct SWBlock {
    WORD    SwNumber; /* Number in SwListArray */
    SWENTRY SwListArray[SwNumber];
}SWBLOCK *SWBLOCKPTR;
```

Where

**SwNumber** is a count of the number of array entries returned.

**SwListArray** is an array with the given number of entries of SWENTRY structures.

The SWENTRY structure is shown below.

### Definition

```
typedef struct swentry {
    HANDLE SwHandle; /* Switch list handle for this entry */
    SWCNTRL SwControl;
} SWENTRY *SWENTRYPTR;
```

Where

**SwHandle** is the Switch List handle for this entry.

**SwControl** is the SWCNTRL structure as given below.

### Definition

```
typedef struct swcontrol {
    char    SwTitle[60+1];
    HANDLE WindowHandle; /* 0 if no window */
    HANDLE IconHandle; /* 0 if no icon */
    HANDLE ProgramHandle /* 0 if not an installed program */
    WORD    ProcessId; /* For this entry */
    WORD    SessionId; /* For this entry */
    BYTE    SwitchFlagA;
    BYTE    SwitchFlagB;
} SWCNTRL *SWCNTRLPTR;
```

Where

**SwTitle** is the ASCIIZ title for this entry. It must be a non-null string in order to define a valid switch list entry.

**WindowHandle** is the handle of the main window belonging to this entry. (0 means that there is no window, in which case this is a non-Presentation Manager entry.)

**IconHandle** is the handle to access the icon for this entry.



**ProgramHandle** is the program handle that was used to start the program resulting in this entry in the switch list. Only items in the switch list that have a program handle entry can be included in the *SAVE CONFIGURATION* function as these are the only programs that the Shell knows how to get started.

**ProcessId** is the DOS process id for this entry.

**SessionId** is the session identifier for this entry. The session id is generated by DOS when starting a new session running. This is a read only field and may NOT be changed by the user.

**SwitchFlagA** defines the visibility of an entry in the Switch List. It is a byte with **ONE** of the following values:

NOT_VISIBLE	No entry in the visual list
NOT_SELECTABLE	Visible, but disabled (grayed)
SWL_NORMAL	Visible and enabled (default value == 0)

- **NOT\_VISIBLE** means that this entry will not appear in the visual switch list, but can be selected by the user by using the **Jump-Application** hotkey, or, if it is a Presentation Manager application, making a direct selection on one of the windows belonging to that application.
- **NOT\_SELECTABLE** means that this entry is not enabled for selection. (On a colour screen it will appear **grayed**.)
- **SWL\_NORMAL** means that this entry is both visible and enabled for selection. (This is the default when 0 is given.)

**SwitchFlagB** defines the response of this entry to the **Jump-Application** hotkey. It is a byte with **ONE** of the following values:

JUMP_ENABLED	Will appear in the round robin sequence
JUMP_DISABLED	Will not appear in the round robin sequence

- **JUMP\_ENABLED** means that this entry will be activated as a result of pressing the **Jump-Application** hotkey.
- **JUMP\_DISABLED** means that this entry will not be activated as a result of pressing the **Jump-Application** hotkey. The application can still be activated by directly selecting the entry in the Switch List (if the entry is visible and enabled) or, if it is a Presentation Manager application, by making a direct selection on one of the windows belonging to that application.

Returns:

```
IF ERROR ( AX = 0 )
ELSE      AX = Total number of switch list entries
```

Remarks

The Switch List Block is built into the block of storage provided by the caller. The total number of Switch List entries will always be returned to the caller.

```
int WSHRemoveSwitchEntry( (HANDLE) SwHandle)
HANDLE SwHandle          /* Switch list handle */
```

---

Purpose Remove an entry from the switch list.

Where: **SwHandle** is the switch list handle of the entry to be removed.

Returns:

```
IF ERROR ( AX not = 0 )
      AX = Error Code
```

- Entry cannot be removed

Remarks

Well behaved applications should call WSHAddSwitchEntry to include themselves in the switch list when they start to run and also call WSHRemoveSwitchEntry before terminating to remove their entry from the switch list.

In addition the window management will ensure that the switch list entry is removed when a window is destroyed by calling WSHRemoveSwitchEntry.

Entries for non-Presentation Manager programs CANNOT be explicitly removed by the program. They will be removed by the Shell when the screen group terminates.

```
int WSHSwitchToProgram( (HANDLE) SwHandle)
HANDLE SwHandle        /* Switch list handle */
```

---

Purpose Make a specific program the active program. This will involve jumping to another window within the Presentation Manager screen group, and bringing that, and all related windows, to the “front” of the screen, or switching to another screen group in the case of a non-Presentation Manager program. In either case, the keyboard (and mouse for non-Presentation Manager) input will be directed to the new program.

Where: **SwHandle** is the switch list entry of the program which is to be made the foreground process.

Returns:

IF ERROR ( AX not = 0 )

AX = Error Code

- Invalid handle
- Requesting program is NOT the foreground process

Remarks

It is only possible to make an arbitrary application the foreground process if you are currently the foreground process. In all other cases the call is ignored. A foreground process is defined as being any process within the active screen group in the case of non-Presentation Manager applications, and the window with the input focus for applications running in the Presentation Manager screen group.

### 3.2.8 Presentation Manager Initialization File and Control Panel API

#### 3.2.8.1 Overview of control panel

The Control Panel allows the user to tailor certain aspects of the system for personal preference, and to alter the system configuration. The functions provided by the control panel allow the user to:

- Set the system clock (date and time)
- Change the screen colors (for example, Scroll bars, title line, window text, etc...)
- Change the default printer
- Change the printer configuration (meaning the port the printer is attached to)
- Change the country settings (Country, time format, number representation, date format, and currency symbol)
- Change the mouse buttons (swap left and right)
- Set the double click time

All of the above functions are interactive, and the user is allowed to try (or look at) the new settings before confirming the change. After the change is made the new settings are permanently stored.

Some of the information above is provided by system calls and are outside the scope of the Shell API. The calls detailed below allow the visual control panel to access data that is either available in the system but not provided in a convenient form, or not available anywhere else.

The information is stored in PRESSERV.INI in a structured binary form. The API calls below access the relevant parts of this structured file. The basic operation done by the API calls will be to execute WINGetProfileString for a section of PRESSERV.INI and then extract the data from the binary data read from the file. Similarly the write operations will construct a binary structure with the information passed across the API and write the new structure to PRESSERV.INI by calling WINWriteProfileString

Listed in the section, "Presentation Manager Initialization File Functions", are the low-level calls used to access PRESSERV.INI.

### 3.2.9 Presentation Manager Initialization File Functions

This section describes the functions used to read from and write to the Presentation Manager initialization file, PRESSERV.INI. The Presentation Manager initialization file is a special MS OS/2 file that contains keyname-value pairs (within specific application sections) that represent runtime options for applications.

It is a binary file and is thus NOT considered a text editable file. Any application wishing to use PRESSERV.INI should use the API given below to read and write the file. In this way users of the file can be shielded from the way in which the file is stored.

There are the following functions:

```
WINQueryProfileInt  
WINQueryProfileString  
WINWriteProfileString  
WSHQueryProfileSize  
WSHQueryProfileData  
WSHWriteProfileData
```

```
int WINQueryProfileInt( (HANDLE)hab, (LPSTR)lpApplicationName,  
                        (LPSTR)lpKeyName, (int)nDefault)  
HANDLE    hab;  
LPSTR     lpApplicationName;  
LPSTR     lpKeyName;  
int       nDefault;
```

---

**Purpose** This function retrieves the value of an integer key from the the Presentation Manager initialization file PRESSERV.INI. The function searches the Presentation Manager initialization file for a key matching the name specified by lpKeyName under the application heading specified by lpApplicationName.

An integer entry in the Presentation Manager initialization file must have the following form:

```
[ <Application-Name> ]  
    <Key-Name> = <value>
```

where value is the key's integer value.

#### Parameters

---

##### Parameter

##### Significance

hab        The value returned by the WinInitialize function.

##### lpApplicationName

A long pointer to a character string naming the application. The string must be a null-terminated ASCII string.

##### lpKeyName

A long pointer to a character string naming a key. The string must be a null-terminated ASCII string.

##### nDefault

A short integer value specifying the default value for the given key if the key cannot be found in the Presentation Manager initialization file.

#### Return Value

The return value is a short integer value specifying the value of the given key if the key exists. Otherwise, it is equal to nDefault.

The WINQueryProfileInt function returns zero, instead of the default value, if the value corresponding to the specified keyname is not an integer. If the value corresponding to the keyname consists of digits followed by non-numeric characters, the function returns the value of the digits. For example, if the entry 'KeyName=102abc' is accessed, the function returns 102.

```
int WINQueryProfileString( (HAB) hab, (LPSTR) lpApplicationName,  
                           (LPSTR) lpKeyName, (LPSTR) lpDefault, (LPSTR) lpReturnedString,  
                           (int) nSize)  
HAB hab;  
LPSTR lpApplicationName;
```

```
LPSTR lpKeyName;  
LPSTR lpDefault;  
LPSTR lpReturnedString;  
int  nSize;
```

---

**Purpose** This function copies a character string from the user profile, PRESSERV.INI, into the buffer pointed to by lpReturnedString. The function searches the Presentation Manager initialization file for a key matching the name specified by lpKeyName under the application heading specified by lpApplicationName. If the key is found, the corresponding string is copied to the buffer. If the key does not exist, the default character string, lpDefault, is copied.

A string entry in the Presentation Manager initialization file must have the following form:

```
[ <Application-Name> ]  
    <Key-Name> = <string>
```

where string is an ASCII string.

If lpApplicationName is NULL, WINQueryProfileString enumerates all application names in PRESSERV.INI and fills the location pointed to by lpReturnedString with a list of application names (not keynames or values). Each application name in the list is terminated with a null character. The last string in the list is terminated with two null characters. WINQueryProfileString returns the length of the list, up to, but not including, the final null.

If lpKeyName is NULL, WINQueryProfileString enumerates all keynames associated with lpApplicationName by filling the location pointed to by lpReturnedString with a list of keynames (not values). Each keyname in the list is terminated with a null character. The last string in the list is terminated with two null characters. WinQueryProfileString returns the length of the list, up to, but not including, the final null.

---

#### Parameters

Parameter	Significance
hab	The value returned by the WinInitialize function.
lpApplicationName	A long pointer to a character string naming the application. The string must be a null-terminated ASCII string.

**lpKeyName**

A long pointer to a character string naming a key. The string must be a null-terminated ASCII string.

**lpDefault**

A long pointer to the character string to be used if the given key does not exist. It must be a null-terminated ASCII string.

**lpReturnedString**

A long pointer to the buffer to receive the character string.

**nSize**

A short integer value specifying the maximum number of bytes to be copied to the buffer. If the actual string is longer, it is truncated.

**Return Value**

The return value is a short integer value specifying the actual number of characters copied to lpReturnedString.

**Notes**

WinQueryProfileString is not case-dependent, so the strings in lpApplicationName and lpKeyName may be in any combination of uppercase and lowercase letters.

```
BOOL WINAPI WriteProfileString( (HAB) hab, (LPSTR) lpApplicationName,  
                               (LPSTR) lpKeyName, (LPSTR) lpString)  
HAB hab;  
LPSTR lpApplicationName;  
LPSTR lpKeyName;  
LPSTR lpString;
```

---

**Purpose**

This function copies the character string pointed to by lpString into the Presentation Manager initialization file, PRESSERV.INI. This function searches the Presentation Manager initialization file for the key named by lpKeyName under the application heading specified by lpApplicationName. If there is no match, it adds a new string entry to the user profile. If there is a matching key, the function replaces that key's value with lpString.

If there is no application field for lpApplicationName, this function creates a new application field and places an appropriate key-value line in that field of the Presentation Manager initialization file.

A string entry in the Presentation Manager initialization file has the following form:

```
[ <Application-Name> ]  
    <Key-Name> = <string>
```

where string is an ASCII string.

Parameters

---

Parameter	Significance
hab	The value returned by the WinInitialize function.
lpApplicationName	A long pointer to a character string naming the application. The string must be a null-terminated ASCII string.
lpKeyName	A long pointer to a character string naming the desired key. The string must be a null-terminated ASCII string.
lpValue	A long pointer to the string to be copied to the file. The string must be a null-terminated ASCII string.

## Return Value

The return value, a Boolean value, is nonzero if the function is successful. Otherwise, it is zero.

**Notes** Applications that make changes to the PRESSERV.INI file in sections that are also accessed by other applications must send a WM\_WININICHANGE message to all applications in the system.

```
int WSHQueryProfileSize( (HANDLE)hab, (LPSTR)lpApplicationName,  
                        (LPSTR)lpKeyName, (int far *)lpValueSize)  
HANDLE    hab;  
LPSTR     lpApplicationName;  
LPSTR     lpKeyName;  
int far * lpValueSize;
```

---

**Purpose** This function returns the size of a particular keyname-value pair in the Presentation Manager initialization file PRESSERV.INI. The function searches the Presentation Manager initialization file for a key matching the name specified by lpKeyName under the application heading specified by lpApplicationName. The size returned includes any trailing zero bytes included for zero terminated strings.

**Where:** **hab** is the anchor block handle as returned by WinInitialize.

**lpApplicationName** is a pointer to an ASCIIZ string identifying the application section within PRESSERV.INI that is of interest.

**lpKeyName** is a pointer to an ASCIIZ string identifying the keyname pair for which the value length is required.

**lpValueSize** is a pointer to an integer value which will be



set to the size of the value field. If an error occurs the contents of this variable will be unchanged.

Returns:

The possible error conditions are

- Application name not found
- Key name not found
- Error accessing PRESSERV.INI

Remarks

This call is provided so that an application can find out the size of a value field in PRESSERV.INI, where that size is unknown, in order to pass the correct buffer size when calling WSHQueryProfileData.

```
int WSHQueryProfileData( (HANDLE)hab, (LPSTR)lpApplicationName,  
                        (LPSTR)lpKeyName, (LPSTR)lpBuffer, (int far *)lpnSize)  
HANDLE      hab;  
LPSTR       lpApplicationName;  
LPSTR       lpKeyName;  
LPSTR       lpBuffer;  
int far *   lpnSize;
```

---

**Purpose** This function returns a string of binary data from the Presentation Manager initialization file PRESSERV.INI. The function searches the Presentation Manager initialization file for a key matching the name specified by lpKeyName under the application heading specified by lpApplicationName.

**Where:** **hab** is the anchor block handle as returned by WINInitialize.  
**lpApplicationName** is the application name identifying the section within PRESSERV.INI.

**lpKeyName** is the keyname identifying the keyname-value pair within PRESSERV.INI.

**lpBuffer** is where the data will be returned into. The returned string will not be zero terminated, unless the value string is explicitly zero terminated within PRESSERV.INI. This is a call handling binary data.

**lpnSize** is a long pointer to an integer value giving the maximum size of the buffer. If the function is successful this will be overwritten with the number of bytes copied into the buffer.

**Returns** If Retcode  $\neq 0$  an error occurred. (Otherwise lpnSize is the number of bytes copied into the buffer.) The possible return code values are

- No match on application name
- No match on key name
- Not enough room for data
- Error accessing PRESSERV.INI

Remarks

Because of the binary nature of the data the returned data is not zero terminated. It is up to the application to use the length provided in order to process the data.

```
int WSHWriteProfileData( (HANDLE)hab, (LPSTR)lpApplicationName,
                        (LPSTR)lpKeyName, (LPSTR)lpBuffer, (int)nSize)
HANDLE    hab;
LPSTR     lpApplicationName;
LPSTR     lpKeyName;
LPSTR     lpBuffer;
int       nSize;
```

---

**Purpose** This function writes a string of binary data of length **nSize** to the Presentation Manager initialization file, putting the data into the area identified by the **Application Name Key Name** parameter pair.

**Where:** **hab** is the anchor block handle as returned by WINInitialize.  
**lpApplicationName** is the application name identifying the section within PRESSERV.INI.

**lpKeyName** is the keyname identifying the keyname-value pair within PRESSERV.INI.

**lpBuffer** is data to be written. This string is NOT zero terminated, and the length is obtained from the following parameter.

**nSize** is the size of the data to be written.

**Retcode** is the return value. The return value is 0 if the function worked, otherwise is one of the error values listed below.

**Returns** Return code values

- Invalid application name
- Invalid key name
- Error accessing PRESSERV.INI

Remarks

Because of the binary nature of the data, the input data is not zero terminated. The length provided is the only way to identify the length of the data.

## 3.3 Shell API Structure definitions

### 3.3.1 Shell API data structure reference

This section lists all the data structures used by the Presentation Manager Shell API.

#### Definition

```
typedef struct gis {
    short  TotalNumber;
    short  ArrayCount;
    struct ProgramEntry ProgramArray[ArrayCount];
} GISSTRUCT *GISPTR;
```

*Where*

**TotalNumber** is the total count of entries in the selected group.

**ArrayCount** is the number of entries for which there was room in the buffer.

**ProgramArray** is an array of structures, one array element for each program entry within the group. The format of the structure is given below.

#### Definition

```
typedef struct ProgramEntry {
    HANDLE    ProgramHandle;
    PROGTYPE  ProgramType;
    /* Program/Group Handle Flag */
    char      InvisibleFlag;
    /* zero if ENTRY is visible on screen,
       non-zero if hidden */
    char      IconFileName[&maxpathl.+1];
    char      ProgramTitle[60+1];
} PROGRAMENTRY *PROGRAMENTRYPTR;
```

*Where*

**ProgramHandle** is the program/group handle.

**ProgramType** is **PROGRAM** or **PROGRAMGROUP**. In the case of **PROGRAM** then the information as to what type of program this is is included. The type can be Presentation Manager, non-Presentation Manager-Windowed, and non-Presentation Manager-other.

**InvisibleFlag** is **VISIBLE** if this entry appears when this group is displayed by the Shell, or **INVISIBLE** if the entry is not shown in the visible list. It does NOT refer to the visibility status of any windows belonging to this entry.

**IconFileName** is a far pointer to an ASCIIZ filename string which is where the icon definition for this entry can be found. The pointer may be

NULL in which case no icon is defined.

**Program Title** is a character array containing the program title for this entry. The maximum length is 60. No leading or trailing blanks are preserved by the Shell API.

### Definition

```
typedef struct progarray {
    short    TotalCount;    /* Total number of entries in list */
    short    ArrayCount;    /* Number of array elements */
    HANDLE   ProgramArr[ArrayCount];    /* Program handles */
} PROGARRAY *PROGARRAYPTR;
```

*Where*

**TotalCount** is the total number of program or program group entries in the list.

**ArrayCount** is the size of the array that fits into the buffer of the given length.

**ProgramArr** is the array of program, or program group handles.

### Definition

```
typedef struct pib {
    PROGTYPED   ProgramType;
                /* Presentation Manager, non-Presentation Manager-Windowed,
                non-Presentation Manager-other,
                Group, and Visibility attribute */
    char        ProgramTitle[60+1];
    char        IconFileName[&maxpath1.+1];
    char        ExecutableName[&maxpath1.+1];
    char        StartupDirectory[&maxpath1.+1];
    XYWINSIZE   InitialPosition;
    char        HelpString[&helpstr1.+1];
    short       EnvironLength;
    char        EnvironString[EnvironLength];
    short       ParameterLength;
    char        ParameterString[1];
} PIBSTRUCT *PIBSTRUCTPTR;
```

*Where*

**ProgramType** defines the type and visibility of this program. If this is a PROGRAM then the type can also be Presentation Manager, non-Presentation Manager-Windowed, and non-Presentation Manager-other. If this is a PROGRAMGROUP, then the type of program has no meaning. The Visibility attribute defines whether this entry is visible in the Start-A-Program list.

**ProgramTitle** is the title for this program. No leading or trailing blanks are preserved by the Shell API.

**IconFileName** defines the icon file associated with this program.

**ExecutableFileName** defines the executable file that will be run when this program is started.

**StartupDirectory** defines the subdirectory that will be the current drive and directory when the program starts running.

**InitialPosition** is the suggested position and size to be used on the first `WINCreateWindow` call. If all values are 0, then the Shell will provide an initial size and position.

**HelpString** is a short informative piece of help information for this program. This text will be displayed whenever general help is requested for this program.

**EnvironLength** is the length of the **EnvironString**.

**EnvironString** defines the environment variables to be passed to the program when it is started. This string is in the format required by DOS, i.e. a set of ASCII strings, the complete set of which is terminated by a null character.

**ParameterLength** is the length of the **ParameterString**.

**ParameterString** defines the parameter to be passed to the program, via its "Command Line" when it is invoked.

Each entry in `ParameterArray` is a structure, as follows.

#### Definition

```
typedef struct parameterdescriptor {
    WORD    ParameterType;
    BYTE    ParameterSubtype;
    short   PstringLength;
    char    ParameterString[PstringLength];
           /* Initial path for FileSystem
              when User Selected File Name
              OR default input string */
} PARAMETERDESCRIPTOR *PARAMETERDESCRIPTORPTR;
```

**ParameterType** is one of three basic types.

- File name

When `ParameterSubtype` will be one of the following

- User selected - by definition, must exist
- User entered - must exist
- User entered - must not exist
- User entered - may or may not exist

- Free format

- Constant

**ParameterSubtype** is one of the defined filename subtypes if **ParameterType** is **File-Name**, otherwise it is undefined.

**PStringlength** is the length of **ParameterString** excluding the terminating zero. This is the length before any substitutions have taken place.

**ParameterString** is an ASCIIZ input string for the program. The meaning of **ParameterString** depends on **ParameterType**. These meanings are shown below.

---

Type	Meaning of ParameterString
constant	Input Parameter for the program.
user selected filename	Initial path to use for file selection when calling the FileSystem.
free format	String from which the input parameter string is constructed, which may include prompting the user for the actual parameter.

### Definition

```
typedef struct ProgramEntry {
    HANDLE    ProgramHandle;
    PROGTYPE  ProgramType;
    /* Program/Group Handle Flag */
    char      InvisibleFlag;
    /* zero if ENTRY is visible on screen,
       non-zero if hidden */
    char      IconFileName[&maxpath1.+1];
    char      ProgramTitle[60+1];
} PROGRAMENTRY *PROGRAMENTRYPTR;
```

*Where*

**ProgramHandle** is the program/group handle.

**ProgramType** is **PROGRAM** or **PROGRAMGROUP**. In the case of **PROGRAM** then the information as to what type of program this is is included. The type can be Presentation Manager, non-Presentation Manager-Windowed, and non-Presentation Manager-other.

**InvisibleFlag** is **VISIBLE** if this entry appears when this group is displayed by the Shell, or **INVISIBLE** if the entry is not shown in the visible list. It does NOT refer to the visibility status of any windows belonging to this entry.

**IconFileName** is a far pointer to an ASCIIZ filename string which is where the icon definition for this entry can be found. The pointer may be NULL in which case no icon is defined.

**Program Title** is a character array containing the program title for this entry. The maximum length is 60. No leading or trailing blanks are preserved by the Shell API.

### Definition

```
typedef struct swcontrol {
    char    SwTitle[60+1];
    HANDLE  WindowHandle;          /* 0 if no window */
    HANDLE  IconHandle;            /* 0 if no icon */
    HANDLE  ProgramHandle          /* 0 if not an installed program */
    WORD    ProcessId;             /* For this entry */
    WORD    SessionId;            /* For this entry */
    BYTE    SwitchFlagA;
    BYTE    SwitchFlagB;
} SWCNTRL *SWCNTRLPTR;
```

### Where

**SwTitle** is the ASCIIZ title for this entry. It must be a non-null string in order to define a valid switch list entry.

**WindowHandle** is the handle of the main window belonging to this entry. (0 means that there is no window, in which case this is a non-Presentation Manager entry.)

**IconHandle** is the handle to access the icon for this entry.

**ProgramHandle** is the program handle that was used to start the program resulting in this entry in the switch list. Only items in the switch list that have a program handle entry can be included in the *SAVE CONFIGURATION* function as these are the only programs that the Shell knows how to get started.

**ProcessId** is the DOS process id for this entry.

**SessionId** is the session identifier for this entry. The session id is generated by DOS when starting a new session running. This is a read only field and may NOT be changed by the user.

**SwitchFlagA** defines the visibility of an entry in the Switch List. It is a byte with **ONE** of the following values:

NOT_VISIBLE	No entry in the visual list
NOT_SELECTABLE	Visible, but disabled (grayed)
SWL_NORMAL	Visible and enabled (default value == 0)

- **NOT\_VISIBLE** means that this entry will not appear in the visual switch list, but can be selected by the user by using the **Jump-Application** hotkey, or, if it is a Presentation Manager application, making a direct selection on one of the windows

belonging to that application.

- **NOT\_SELECTABLE** means that this entry is not enabled for selection. (On a colour screen it will appear **grayed**.)
- **SWL\_NORMAL** means that this entry is both visible and enabled for selection. (This is the default when 0 is given.)

**SwitchFlagB** defines the response of this entry to the **Jump-Application** hotkey. It is a byte with **ONE** of the following values:

<b>JUMP_ENABLED</b>	Will appear in the round robin sequence
<b>JUMP_DISABLED</b>	Will not appear in the round robin sequence

- **JUMP\_ENABLED** means that this entry will be activated as a result of pressing the **Jump-Application** hotkey.
- **JUMP\_DISABLED** means that this entry will not be activated as a result of pressing the **Jump-Application** hotkey. The application can still be activated by directly selecting the entry in the Switch List (if the entry is visible and enabled) or, if it is a Presentation Manager application, by making a direct selection on one of the windows belonging to that application.

## Definition

```
typedef struct SWBlock {  
    WORD      SwNumber; /* Number in SwListArray */  
    SWENTRY SwListArray[SwNumber];  
}SWBLOCK *SWBLOCKPTR;
```

*Where*

**SwNumber** is a count of the number of array entries returned.

**SwListArray** is an array with the given number of entries of **SWENTRY** structures.

The **SWENTRY** structure is shown below.

## Definition

```
typedef struct swentry {  
    HANDLE SwHandle; /* Switch list handle for this entry */  
    SWCNTRL SwControl;  
} SWENTRY *SWENTRYPTR;
```

*Where*

**SwHandle** is the Switch List handle for this entry.

**SwControl** is the **SWCNTRL** structure as given below.

## Definition



```
typedef struct swcontrol {
    char    SwTitle[60+1];
    HANDLE  WindowHandle;      /* 0 if no window */
    HANDLE  IconHandle;        /* 0 if no icon */
    HANDLE  ProgramHandle     /* 0 if not an installed program */
    WORD    ProcessId;        /* For this entry */
    WORD    SessionId;        /* For this entry */
    BYTE    SwitchFlagA;
    BYTE    SwitchFlagB;
} SWCNTRL *SWCNTRLPTR;
```

Where

**SwTitle** is the ASCIIZ title for this entry. It must be a non-null string in order to define a valid switch list entry.

**WindowHandle** is the handle of the main window belonging to this entry. (0 means that there is no window, in which case this is a non-Presentation Manager entry.)

**IconHandle** is the handle to access the icon for this entry.

**ProgramHandle** is the program handle that was used to start the program resulting in this entry in the switch list. Only items in the switch list that have a program handle entry can be included in the *SAVE CONFIGURATION* function as these are the only programs that the Shell knows how to get started.

**ProcessId** is the DOS process id for this entry.

**SessionId** is the session identifier for this entry. The session id is generated by DOS when starting a new session running. This is a read only field and may NOT be changed by the user.

**SwitchFlagA** defines the visibility of an entry in the Switch List. It is a byte with **ONE** of the following values:

NOT_VISIBLE	No entry in the visual list
NOT_SELECTABLE	Visible, but disabled (grayed)
SWL_NORMAL	Visible and enabled (default value == 0)

- **NOT\_VISIBLE** means that this entry will not appear in the visual switch list, but can be selected by the user by using the **Jump-Application** hotkey, or, if it is a Presentation Manager application, making a direct selection on one of the windows belonging to that application.
- **NOT\_SELECTABLE** means that this entry is not enabled for selection. (On a colour screen it will appear **grayed**.)
- **SWL\_NORMAL** means that this entry is both visible and enabled for selection. (This is the default when 0 is given.)

**SwitchFlagB** defines the response of this entry to the **Jump-Application** hotkey. It is a byte with **ONE** of the following values:

**JUMP\_ENABLED** Will appear in the round robin sequence  
**JUMP\_DISABLED** Will not appear in the round robin sequence

- **JUMP\_ENABLED** means that this entry will be activated as a result of pressing the **Jump-Application** hotkey.
- **JUMP\_DISABLED** means that this entry will not be activated as a result of pressing the **Jump-Application** hotkey. The application can still be activated by directly selecting the entry in the Switch List (if the entry is visible and enabled) or, if it is a Presentation Manager application, by making a direct selection on one of the windows belonging to that application.

### Definition

```
typedef struct swentry {  
    HANDLE SwHandle;           /* Switch list handle for this entry */  
    SWCNTRL SwControl;  
} SWENTRY *SWENTRYPTR;
```

Where

**SwHandle** is the Switch List handle for this entry.

**SwControl** is the SWCNTRL structure as given below.

### Definition

```
typedef struct swcontrol {  
    char SwTitle[60+1];  
    HANDLE WindowHandle;      /* 0 if no window */  
    HANDLE IconHandle;        /* 0 if no icon */  
    HANDLE ProgramHandle      /* 0 if not an installed program */  
    WORD ProcessId;           /* For this entry */  
    WORD SessionId;           /* For this entry */  
    BYTE SwitchFlagA;  
    BYTE SwitchFlagB;  
} SWCNTRL *SWCNTRLPTR;
```

Where

**SwTitle** is the ASCIIZ title for this entry. It must be a non-null string in order to define a valid switch list entry.

**WindowHandle** is the handle of the main window belonging to this entry. (0 means that there is no window, in which case this is a non-Presentation Manager entry.)

**IconHandle** is the handle to access the icon for this entry.

**ProgramHandle** is the program handle that was used to start the program resulting in this entry in the switch list. Only items in the switch list that have a program handle entry can be included in the *SAVE CONFIGURATION* function as these are the only programs that the Shell knows how to get started.

**ProcessId** is the DOS process id for this entry.

**SessionId** is the session identifier for this entry. The session id is generated by DOS when starting a new session running. This is a read only field and may NOT be changed by the user.

**SwitchFlagA** defines the visibility of an entry in the Switch List. It is a byte with **ONE** of the following values:

NOT_VISIBLE	No entry in the visual list
NOT_SELECTABLE	Visible, but disabled (grayed)
SWL_NORMAL	Visible and enabled (default value == 0)

- **NOT\_VISIBLE** means that this entry will not appear in the visual switch list, but can be selected by the user by using the **Jump-Application** hotkey, or, if it is a Presentation Manager application, making a direct selection on one of the windows belonging to that application.
- **NOT\_SELECTABLE** means that this entry is not enabled for selection. (On a colour screen it will appear **grayed**.)
- **SWL\_NORMAL** means that this entry is both visible and enabled for selection. (This is the default when 0 is given.)

**SwitchFlagB** defines the response of this entry to the **Jump-Application** hotkey. It is a byte with **ONE** of the following values:

JUMP_ENABLED	Will appear in the round robin sequence
JUMP_DISABLED	Will not appear in the round robin sequence

- **JUMP\_ENABLED** means that this entry will be activated as a result of pressing the **Jump-Application** hotkey.
- **JUMP\_DISABLED** means that this entry will not be activated as a result of pressing the **Jump-Application** hotkey. The application can still be activated by directly selecting the entry in the Switch List (if the entry is visible and enabled) or, if it is a Presentation Manager application, by making a direct selection on one of the windows belonging to that application.

### Definition

```
typedef struct xywinsize {
    short  XPos;      /* X position of Window */
    short  YPos;      /* Y position of Window */
    short  XSize;     /* X extent of Window */
    short  YSize;     /* Y extent of Window */
    word   WinFlag;   /* MINIMISED or MAXIMISED
                      or INVISIBLE or NORMAL (0) */
} XYWINSIZE *XYWINSIZEPTR;
```

1

2

3

# Chapter 4

## Window Management Functions

---

4.1	Window management functions	159
4.1.1	Window Manager Architecture	159
4.1.1.1	The Window	159
4.1.1.2	Window Procedures and Window Messages	159
4.1.1.3	Window Classes and Instances	160
4.1.1.4	Public and Private Window Classes	161
4.1.1.5	Window Attributes	161
4.1.1.6	Window Relationships	163
4.1.1.7	Window Owners and Owned Windows	165
4.1.1.8	Window States	165
4.1.1.9	Object Windows	165
4.1.1.10	Window Subclassing	166
4.1.1.11	Special Windows	166
4.1.1.12	Window Management Routines	167
4.1.2	Window Drawing Management Architecture	209
4.1.2.1	The Window DC	209
4.1.2.2	Cached Micro-PS	210
4.1.2.3	Application PS vs. Cache PS Considerations	210
4.1.2.4	Window Clipping Options	211
4.1.2.5	Window Clipping Considerations	212
4.1.2.6	Application PS Example	213
4.1.2.7	Cached-PS Example	213
4.1.2.8	Window Repainting after Window Rearrangement	214
4.1.2.9	Synchronous Window Updating	215
4.1.2.10	Synchronous vs. Asynchronous Painting	216
4.1.3	Window Drawing Functions	216

4.1.3.1	Drawing Helpers	224
4.1.4	Window Frames	230
4.1.4.1	Window Frame Architecture	230
4.1.4.2	The Frame Window Class	232
4.1.4.3	Standard Window Frame Routines	238
4.1.4.4	Using Frame Windows	242
4.1.4.5	Alternate Window Frame Formatting	242
4.1.5	The Title Bar Control	242
4.1.5.1	Title Bar Style	243
4.1.5.2	Title Bar Messages	243
4.1.5.3	Title Bar Notification Messages	245
4.1.6	The Size Control	245
4.1.6.1	Size Control Styles	246
4.1.6.2	Size Control Messages	246
4.1.6.3	Size Notification Codes	246
4.1.7	The Minimize/Maximize Control	246
4.1.7.1	MinMax Control Styles	247
4.1.8	Dialog Boxes	247
4.1.8.1	The Dialog Procedure	248
4.1.8.2	Dialog Templates	249
4.1.8.3	Dialog Control Groups	252
4.1.8.4	Dialog Box Messages	252
4.1.8.5	Dialog Styles	254
4.1.8.6	Dialog Box Routines	254
4.1.9	Message Boxes	261
4.1.9.1	Message Box Functions	261
4.1.10	Control Windows	266
4.1.10.1	Common Features	267
4.1.10.2	Standard Control Messages	270
4.1.10.3	Owner Notification Messages	273
4.1.11	Static Controls	276

4.1.11.1	Static Control Styles	276
4.1.11.2	Static Notification Codes	278
4.1.12	Button Controls	278
4.1.12.1	Button Control Styles	278
4.1.12.2	Button Control Messages	279
4.1.12.3	Button Notification Codes	280
4.1.12.4	Button Control State Messages	282
4.1.13	Edit Controls	283
4.1.13.1	Edit Control Styles	284
4.1.13.2	Edit Control Keys	284
4.1.13.3	Edit Control Notification Messages	285
4.1.13.4	Edit Control Messages	286
4.1.14	Listbox Controls	289
4.1.14.1	Listbox Control Styles	289
4.1.14.2	Listbox Control Notification Messages	290
4.1.14.3	Listbox Control States	291
4.1.14.4	Listbox Control Messages	291
4.1.15	Scroll Bar Controls	297
4.1.15.1	Scroll Bar Notification Messages	298
4.1.16	Menu Controls	302
4.1.16.1	Action Bar Layout	302
4.1.16.2	Menu Control Styles	303
4.1.16.3	Menu Items	304
4.1.16.4	Menu Item Styles	304
4.1.16.5	Menu Item Attributes	306
4.1.16.6	Mnemonics and Mnemonic Highlighting.	306
4.1.16.7	Menu Data Structures	307
4.1.16.8	Menu Notification Messages	309
4.1.16.9	Menu Control Messages	311
4.1.17	Caret Manager	318
4.1.18	Mouse Cursor	321

4.1.19	Clipboard Manager	325
4.1.20	Rectangle Functions	337
4.1.21	Presentation Manager Resources	343
4.1.22	Command Key Accelerators	343
4.1.23	System Colors	350
4.1.24	System Information Functions	352
4.1.25	Using Windows of Other Threads	356
4.1.26	Window Destroy Registration	359
4.1.27	System and Queue Hooks	360
4.1.28	International Information	368
4.1.29	Miscellaneous	376



## 4.1 Window management functions

### 4.1.1 Window Manager Architecture

#### 4.1.1.1 The Window

A window is the standard input and output tool of a Presentation Manager application program. Some Presentation Manager applications use a window to display text and graphics on the system screen and to process input from the system keyboard, mouse, and timer. In a virtual sense, a window provides the same input and output capabilities as a complete graphics terminal, but it does so without requiring complete control of the system's actual hardware resources.

Windows are identified by a window handle which uniquely identifies the window.

All windows in the system are associated with a particular message queue. A message queue is associated with a particular thread; a thread may have only one message queue. Windows are always associated with the message queue that was associated with the thread that created with the window. More than one window can be associated with a particular message queue.

Though there is an association between a window and its message queue thread, a window handle may be accessed by all threads and processes.

For more information on Message Queues, see the "Message Manager" section.

#### 4.1.1.2 Window Procedures and Window Messages

Every window in the system has a procedure associated with it called the "Window Procedure". The window procedure, also called the "Window Proc", controls all aspects of a window: what it looks like, how it responds to state changes, how it processes input, etc.

The information passed to a window procedure is called a "Window Message". A window message is made up of five parts, which correspond to the four arguments of the window proc and its return value:

HWND	hwnd	- Handle of window receiving the message
unsigned	msg	- Msg ID identifying the message
ULONG	lParam1	- ULONG parameter (content depends on message ID)
ULONG	lParam2	- ULONG parameter (content depends on message ID)
ULONG	reply	- 32-bit return value (content depends on message ID)

The message ID defines the message. The contents of `lParam1` and `lParam2`, and whether or not a return value is required, depend on the semantics of the message as defined by the message ID.

For more information on window messages and some rules for their use, see the "Message manager" section.

#### 4.1.1.3 Window Classes and Instances

Every window is an "instance" of a particular window "class". The window class defines the window procedure to be used, as well as other information that is shared by all instances of the class. Before an instance of a particular class can be created, the window class must be registered with the window manager.

There are a number of preregistered window classes available to applications, that are used to implement menus, scroll bars, pushbuttons, etc. All of these preregistered classes are explained in detail later.

Window classes are identified by the "class name". Class names are normally specified as far pointers to standard zero-terminated strings. Case is significant in class name strings.

Instead of a far pointer to a string, preregistered class names are specified as a 32-bit value with `0xffff` in the hi word, and a small integer in the low word. The names of these special class name constants begin with "WC\_":

---

Class Name	Description
WC_FRAME	Standard top-level window frame class
WC_DIALOG	Standard dialog box window class
WC_BUTTON	Pushbuttons, checkboxes, and radio buttons
WC_EDIT	Text editing fields
WC_STATIC	Static display items such as text strings and icons.
WC_LISTBOX	Lists of text that the user can choose from.

WC\_MENU

Menus

WC\_SCROLLBAR

Scroll bars

WC\_MINMAXBOX

Minimize/Maximize pushbuttons

WC\_SIZEBORDER

Window sizing border

#### **4.1.1.4 Public and Private Window Classes**

There are two kinds of window classes: public classes and private classes. Applications can create windows of private classes only within the process in which the class was registered. Private window class names need only be unique for that process; more than one application may register private window classes with the same class name.

Public window classes are window classes that can be used in any process context to create window instances. Public class window procedures must be callable from any process/thread context, so they must be defined as part of a dynalink library. Public class names must be unique for all applications; by convention, public class names should contain the module name as part of their name. For example:

```
"ModuleName.ClassName"
```

Window instances created from public or private classes can be used by any process in the system; the "public" and "private" terms refer only to the scope of the window class when used to create a window.

#### **4.1.1.5 Window Attributes**

##### *4.1.1.5.1 Window Style*

The "Window Style" is a 32-bit value that controls the appearance and behavior of a window. The window style is specified by combining various style values together with the OR operator. The meaning and interpretation of the 32 style bits depends on the window class. There are a few standard style bits that apply to all window classes, however. These styles, whose names begin with "WS\_", are restricted to the high order 16 bits of the window style; this leaves the lower 16 bits available for use and interpretation by the window procedure of the window class.

#### *4.1.1.5.2 Class Style*

In addition to window styles, there are styles which apply to all the windows of a given class. These are used in a similar way to Window Styles, except that they are specified when a Window Class is registered and then apply to ALL windows of that class. These styles have names beginning with "CS\_" and are 32-bit values.

#### *4.1.1.5.3 Window Rectangle and coordinates*

All window positions and sizes are specified in "window coordinates". Window coordinates are in pixel sized units, with the origin (0, 0) at the bottom left corner of a window.

Every window has a rectangle associated with it that describes the size and position of the window on the screen. The coordinates of this RECT structure are in window coordinates.

See the section, "Rectangle Functions", for more information on the RECT structure.

#### *4.1.1.5.4 Window ID*

All windows have an ID, which is a 16-bit value that is specified by the application when the window is created. The window ID can be used to refer to the window with that ID.

#### *4.1.1.5.5 Window Text*

Many classes of windows have a string associated with them, called the "Window Text". This string is often displayed in the window.

#### *4.1.1.5.6 Window Words*

For window classes registered by applications, it is possible to reserve extra memory for each window that can be used by the window procedure to store any extra information that may be required for the window class.

When the window class is registered, the number of extra bytes to reserve is specified. Every window instance of that class will have these bytes allocated and initialized to 0, which can be read and modified by the application or window procedure.

The `WinQueryWindowUInt()`, `WinSetWindowUInt()`, `WinQueryWindowULong()`, and `WinSetWindowULong()` functions are the only means for accessing window words.

#### 4.1.1.6 Window Relationships

Main application windows are called "top level" windows. All of these windows are arranged in an overlapping fashion: you cannot see any part of a window overlapped by windows above it.

Windows may also contain other windows. Windows that contain other windows are called "Parent Windows"; the windows within the parent are called "Child windows". Child windows are always clipped to the parent; only the part of a child that lies within the parent's window rectangle are visible.

Windows that share the same parent are called "Sibling Windows". Like top level windows, siblings are arranged in an overlapping fashion. All sibling windows are linked together. The first window on the list is considered the "Top window" of the list; the last window is known as the "Bottom window". Since all siblings are arranged in overlapping order this order represents the Z-axis ordering of the windows.

All children of a window, and their children, etc., are called the "Descendants" of that window.

Below is a diagram illustrating one possible arrangement of some windows on the screen. The next diagram shows the parent/child relationships of these windows.

Notice that the windows are arranged in a tree, with a special window called the "Desktop window" at the top of the tree. Windows A, B, and C are top level windows, which are all children of the desktop window. The window handle of the desktop window is not available to the application; for those functions that require a parent window handle, `NULL` is used to indicate the desktop window.

There are a few important points to note from this diagram. Window J illustrates how windows are always contained within their parent. Windows A, B, and C illustrate how windows are obscured by siblings above them.

The actual window clipping and exclusion rules for windows is described in "Window Drawing Management Architecture".

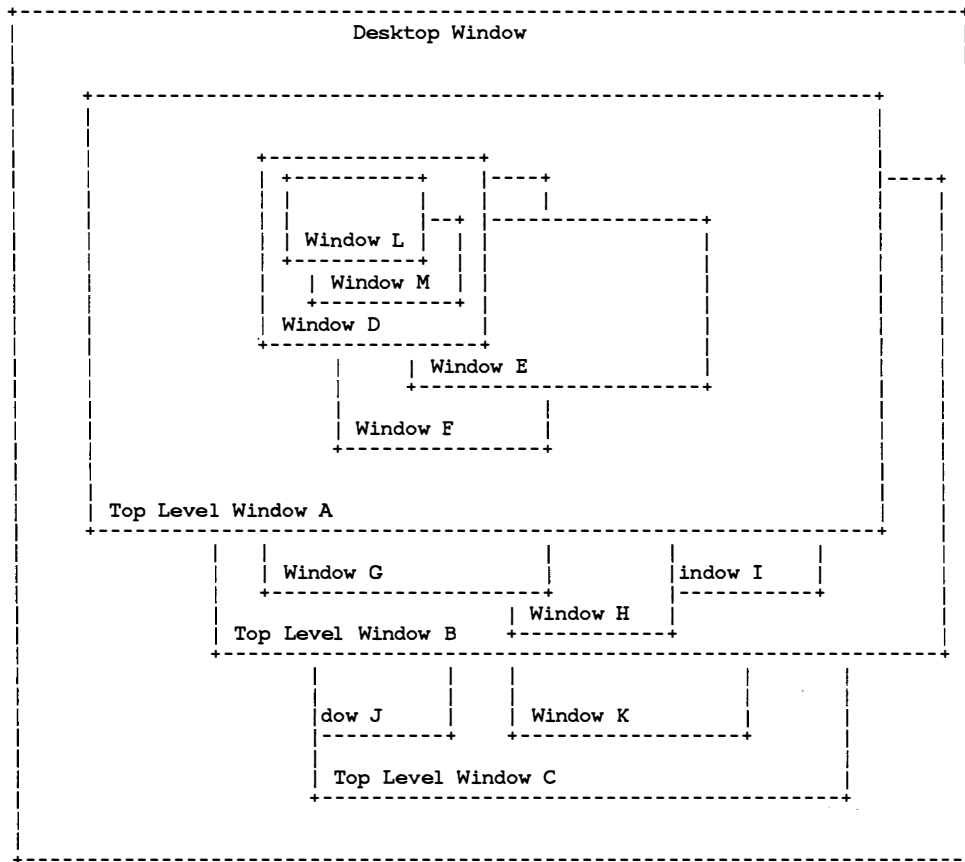
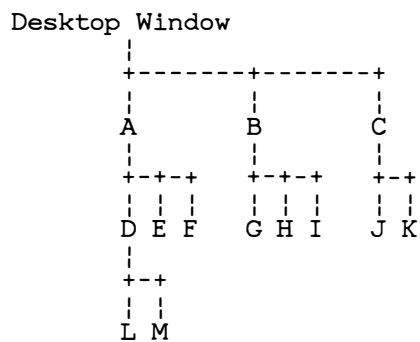


Figure 4.1 Parent and Child Windows



<== Topmost/First      Bottommost/Last ==>

**Figure 4.2 Parent/Child relationships of Previous Diagram**

#### **4.1.1.7 Window Owners and Owned Windows**

When a window is created, an "owner window" may be specified. The window is said to be an "owned window" of the owner window.

The owned and owning relationship between windows may be specified by the application. The effect of owning or being owned by a window depends on whether the windows involved are top level windows or child windows:

- Child windows may send notification messages to their owner windows.
- When a top level window is hidden, minimized, or destroyed, other top level windows owned by that window are hidden or destroyed.

#### **4.1.1.8 Window States**

A window can be visible or invisible. An invisible (or "hidden") window is simply made invisible. Its size and position remain the same, as do its parent and owner windows. However, any drawing in the window is not shown on the screen. The visible state is controlled by the `WS_VISIBLE` window style bit.

Windows can also be disabled. A disabled window is still visible, but it does not respond to mouse input. Windows are normally enabled. The enabled state is controlled by the `WS_ENABLED` window style bit.

Some window classes support other window states in addition to the visible and enabled states. These states are typically changed and queried with window messages.

#### **4.1.1.9 Object Windows**

Windows do not necessarily have to be a parent or child of a window on the screen. Windows that are not part of the desktop window parent tree are called "object windows". They have a window procedure like other windows, and thus can be sent messages. Object windows can also be owned by other windows (either object or normal), and they can have child windows. But, because object windows are not part of the desktop window tree, it can't be made visible on the screen.

Windows whose parent window handles have the value of `HWND_OBJECT` are considered object windows. It is possible to change the parent of a window with `WinSetParent()`. This function can be used to make a normal window an object window by setting its parent to

HWND\_ OBJECT or to make an object window a normal window by setting its parent to another normal window.

Object windows have a size and position like other windows. If the object window is later given a parent window, the position is relative to the top left corner of the new parent.

Object windows are also known as "Orphaned" windows, as they are windows without parents.

#### **4.1.1.10 Window Subclassing**

"Window Subclassing" refers to either modifying the behavior of an instance of a window class, or creating a new window class using an existing window class. In either case, the subclass window procedure is a "hook" into the normal window procedure. The subclass window proc may process any message send to the window, or choose to pass it on to the standard window procedure, or both. In this way it is possible to modify the behavior of a window without rewriting the entire window procedure.

To subclass a window instance, it is simply necessary to replace the window's window procedure with a new window procedure. The previous window procedure is called in place of WinDefWindowProc() by the new window procedure. The WinSubclassWindow() function is used to subclass a window.

To create a subclass from an existing class, it is necessary to register a new window class with a window procedure that calls the existing class window procedure in place of WinDefWindowProc(). The window proc address and other information necessary to register a subclass may be obtained with the WinQueryClassInfo() function.

#### **4.1.1.11 Special Windows**

There are a number of special windows in the system. These special windows generally have to do with the routing of mouse and keyboard input and commands.

##### *4.1.1.11.1 Active Window*

The "Active Window" is the top level window that either has or contains the "focus window" (see below). It is usually the topmost window, and is highlighted in some way. The active window is the window that the user is currently interacting with.



#### *4.1.1.11.2 Focus Window*

The "Focus Window" is the window that will receive all keyboard input. See the "Input Manager" for more information on the focus window.

#### *4.1.1.11.3 Capture Window*

The "Capture Window" is the window that receives all mouse input, even when the mouse is not over the window. There is normally no capture window; in that case mouse input is routed to the window underneath the mouse cursor.

#### *4.1.1.11.4 System Modal Window*

The "System Modal Window" is a special top level window that receives all mouse and keyboard input. Input may also be routed to one of its children. All other top level windows behave as if they are disabled; no interaction is possible.

### **4.1.1.12 Window Management Routines**

#### *4.1.1.12.1 Window Class Styles*

---

Window Class Styles	Meaning
---------------------	---------

CS_SAVEBITS	
-------------	--

	This style controls whether or not the screen image of the area underneath the window is saved when the window is made visible. For more information, see "WinShowWindow()".
--	--

CS_SIZEREDRAW	
---------------	--

	This style determines whether a window should be redrawn when sized. This style should be used for a window whose contents are sensitive to the size of the window. For example, the data in some windows can be scaled up or down to fit the size of the Client Area. In other windows, the data remains the same size whatever the size of the window - it is merely clipped if the window is made smaller. The CS_SIZEREDRAW style should be used in the first case but not in the second. For more information, see "WM_CALCVALIDRECTS".
--	--

CS\_ SYNCPAINT

Window will be synchronously repainted. See "Window Painting After Window Rearrangement" for more information.

CS\_ PARENTCLIP

This style controls how the window is clipped when drawing into it. For more information, see "Window Drawing Management".

4.1.1.12.2 *Standard Window Styles*

---

Standard Window Styles

Meaning

WS\_ VISIBLE

Window is visible. The absence of this bit indicates that a window is invisible. See "WinShowWindow()" for more information.

WS\_ DISABLED

Window is disabled. The absence of this bit indicates that a window is enabled. See "WinEnableWindow()" for more information.

WS\_ CLIPCHILDREN

Causes child windows to be excluded when drawing in the window. Normally, child windows are not excluded. See "Window Drawing Management" for more information.

WS\_ CLIPSIBLINGS

Causes sibling windows to be excluded when drawing in the window. Normally, siblings are not excluded. See "Window Drawing Management" for more information.

WS\_ GROUP

Used to identify the dialog items that make up a "group". See the "Dialog Manager" for more information.

WS\_ TABSTOP

Used to identify dialog items that will be enumerated when the TAB key is pressed. See "Dialog Manager" for more information.

WS\_ SYSMODAL

The window is a system modal window. This means that no other window -- even windows belonging to other applications -- can receive the focus until the window relinquishes control (i.e. is destroyed with WinDestroyWindow). This style is generally used for message boxes and other critical error dialog boxes.

**WS\_MOVE**

A window with this style causes a WM\_MOVE message to be generated by the window manager when it is moved relative to the screen. See the 'WM\_MOVE' message description for more detail.

#### 4.1.1.12.3 Window Class Routines

---

**WinRegisterClass**

---

**Format**

```
BOOL WinRegisterClass(hab, lpszClassName,  
                     lpfnWndProc,  
                     styleClass,  
                     cbWndExtra, idModule);  
  
LPSTR lpszClassName;  
FARPROC lpfnWndProc;  
ULONG styleDefault;  
INT cbWndExtra;  
UINT idModule;  
HAB hab;
```

**Description**

This function registers a window class having the supplied characteristics with the window manager. Returns TRUE if successful, FALSE otherwise. Once registered, the class name may be used to create windows of that class.

lpszClassName is a far pointer to the class name string, or one of the WC\_\* class name constants to specify any of the preregistered classes. The actual class name to be used for the preregistered classes may be found in the description of the class elsewhere in this manual.

lpfnWndProc is the window procedure address to use when creating instances of this class.

styleClass specifies the window class style.  
will

cbWndExtra specifies the number of bytes of window words to reserve when a window is created.

idModule is the module handle that contains the code of the window procedure, returned by the DOS DosLoadModule call. If idModule is NULL, the class is assumed to be a private class. Otherwise, idModule must be a module handle of a dynamic link library containing the code of the

window procedure. The class name must also be unique; this can be achieved by naming the class by prepending the module name:

"ModuleName.ClassName"

Notes     Registering a public class with the same name as an existing public class replaces the existing class. Registering a private class with the same name as another private class of the same process fails and WinRegisterClass() returns FALSE. Applications should avoid registering private classes with the same name as an existing public class. Private classes with the same name as a public class preempt the public class for that process.

See "Public and Private Window Classes".

### WinQueryClassName

#### Format

```
INT WinQueryClassName(HWND hwnd, LPSTR lpchBuffer, cchBufferMax)
HWND hwnd;
LPSTR lpchBuffer;
INT cchBufferMax;
```

#### Description

This function copies the null terminated class name string of the specified window into \*lpchBuffer, returning the length of the string not including the null termination character. If the class name is longer than cchBufferMax, it is truncated at (cchBufferMax - 1) characters; the string is always null terminated.

If the window is of any of the preregistered WC\_\* classes, the string returned will be of the form "#n", where n is a single digit corresponding to the value of the WC\_\* class name constants.

#### *4.1.1.12.4 Creating and Destroying Windows*

```
typedef struct CREATESTRUCT {
    UCHAR FAR *lpPresParams;
    UCHAR FAR *lpCtlData;
    UINT id;
    HWND hwndInsertBehind;
    HWND hwndOwner;
    HWND hwndParent;
    INT cy;
    INT cx;
    INT y;
```

```
    INT x;  
    ULONG style;  
    LPSTR lpszName;  
    LPSTR lpszClass;  
} CREATESTRUCT;
```

---

## WinCreateWindow

---

### Format

```
HWND WinCreateWindow(hab, lpszClassName,  
                    lpszName, style, x, y, cx, cy,  
                    hwndParent, hwndOwner, hwndInsertBehind,  
                    id, lpCtlData, lpPresParams );  
LPSTR lpszClassName;  
LPSTR lpszName;  
ULONG style;  
INT x, y, cx, cy;  
HWND hwndParent;  
HWND hwndOwner;  
HWND hwndInsertBehind;  
UINT id;  
HAB hab;  
UCHAR FAR *lpCtlData;  
UCHAR FAR *lpPresParams;
```

### Description

This function creates a new window, returning the window handle of the window or NULL if unsuccessful.

An instance of the window class named by `lpszClassName` is created. `lpszClassName` may be a far pointer to a registered class name string, or it may be one of the preregistered class names shown in the section, "Window Classes and Instances".

`lpszName` points to the window text, or to other class-specific data. The actual structure of the data pointed to by `lpszName` is class-specific, but it is usually a zero terminated string, which is often displayed in the window.

`style` specifies the style of the window. Any of the standard `WS_` style bits may be used, in addition to any class-specific styles that may be defined for that class.

`x` and `y` specify the position of the window, in window coordinates relative to the origin of the parent window (specified below). `cx` and `cy` are the horizontal and vertical dimensions of the window, also in window coordinate units.

hwndParent specifies the parent window. If hwndParent is NULL, a top level window is created. If hwndParent is HWND\_OBJECT, an object window is created.

hwndOwner is the 'owner' window. In the case of windows with notification codes (like controls), the owner is the window to notify when something happens to the control. When a window gets destroyed, all windows it owns get destroyed also.

hwndInsertBehind specifies the Z-ordering of the newly created window in relation to another window. The new window is placed immediately behind the window specified. hwndInsertBehind can take the values HWND\_TOP and HWND\_BOTTOM to put the new window at the top or bottom of the Z-ordering, respectively.

id is an application-specified value typically used to identify a window. For example, all controls of a dialog must have unique IDs so the owner window when notified can determine which control has notified it. For standard formatting of frame controls, standard IDs must be used identifying each control type. See Frame Formatting.

lpCtldata is a far pointer to class-specific data that may be used to pass information to the window procedure as the window is created. This field is available to the window proc (as are all of the other parameters above) in the CREATESTRUCT structure passed with the WM\_CREATE message. It may also be accessed at any time by the WM\_SETWINDOWPARAMS and WM\_QUERYWINDOWPARAMS messages.

lpPresParams is a reserved field, and must have a value of zero.

#### Notes

A window is normally created enabled and invisible. See the list of the standard window styles for more information on the initial state of a created window.

Messages may be received from other processes or threads when WinCreateWindow() is called.

WinCreateWindow() sends the WM\_CREATE message to the window being created. If the window is created with the WS\_VISIBLE style, WinCreateWindow() will call WinShowWindow(), which may cause additional messages to be sent.

The WM\_SIZE message is NOT sent by WinCreateWindow() while the window is being created. Any required size processing can be performed during the processing of the WM\_CREATE message.

Since windows are often created initially with zero height or width and sized later with a non-zero size, it is a good idea to avoid performing any size-related processing if the size of the window is zero.

It should be noted that there is no limitation to the size and position specified for a window within the number range permitted for the size and position parameters. This means that an application can create windows which are larger than the screen or which are positioned partially or wholly off the screen.

It should be remembered, however, that the user interface for manipulation of window sizes and positions is affected if part or all of the window is off the screen. It is recommended that part of the title bar is left on the screen, if the window has one, to allow the user to move the window around.

These considerations apply to all windows, but particularly to Standard windows.

## WinDestroyWindow

---

### Format

```
BOOL WinDestroyWindow(HWND hwnd)
```

### Description

This function destroys a window and all its descendants. Before the specified window is itself destroyed, all other top level windows owned by hwnd are destroyed. Returns TRUE if successful, FALSE if hwnd is an invalid window handle or is not associated with the current thread.

### Notes

If hwnd is locked, WinDestroyWindow() does not return until the window is unlocked. See the section on "Window Locking".

Messages may be received from other processes or threads.

Messages sent before WinDestroyWindow() returns:

---

Message

When Sent

WM\_DESTROY

Always sent to the window being destroyed after the window has been hidden from the screen, but before its children have been destroyed. The message is sent first to the window being destroyed, then to the children as they are destroyed. Therefore, during the processing of the WM\_DESTROY message, it can be assumed that all the children still exist.

WM\_KILLFOCUS

Sent if the window being destroyed or any of its descendants is the focus window.

WM\_ACTIVATE

Sent with LOUINT(IParam1) == FALSE if the window being destroyed is the active window.

WM\_OTHERWINDOWDESTROYED

Sent to all top level windows if hwnd or any of its descendants has been registered with WinRegisterWindowDestroy(). See "Window Locking".

WM\_RENDERALLFORMATS

Sent if the clipboard owner is being destroyed, and there are unrendered formats in the clipboard. See the "Clipboard Manager".

If the application has associated a presentation space with the window, it must disassociate the PS (using GpiAssociate(hgpi, NULL) ) before calling WinDestroyWindow(). If it does not do so, WinDestroyWindow returns an error. Alternatively, the application can destroy the PS before it destroys the window.

---

WM\_CREATE

---

Format

WM\_CREATE

lParam1: BYTE FAR \*lpCtlData;  
 lParam2: CREATESTRUCT FAR \*lpCreateStruct;  
 Returns: BOOL fError;



**Description**

This message is sent from WinCreateWindow(). The app uses this message for any window initialization that needs to be done. The window procedure returns TRUE to indicate that some error occurred, and the window should not be created (i.e., WinCreateWindow() should return NULL), and FALSE to indicate that creation should proceed normally (and the window should be created).

lparam1 contains the Control Data parameter which is passed to WinCreateWindow(). lparam2 contains a far pointer to a CREATESTRUCT structure, whose fields correspond to the parameters of WinCreateWindow().

---

**WM\_DESTROY**

---

**Format**

```
WM_DESTROY
lParam1:    0;
lParam2:    OL;
Returns:    OL;
```

**Description**

This message is sent when WinDestroyWindow() is called, after the window has been removed from the screen. The WM\_DESTROY message is used to destroy window instance related data, before the window structure has been freed.

#### *4.1.1.12.5 Enabling or Disabling a Window*

---

**WinEnableWindow**

---

**Format**

```
BOOL WinEnableWindow(hwnd, fEnable)
HWND hwnd;
BOOL fEnable;
```

**Description**

This routine is used to enable or disable a window. If fEnable is TRUE, hwnd is enabled, otherwise hwnd is disabled. The enabled state is changed by setting or clearing the WS\_DISABLED window style bit, as appropriate. Returns TRUE if the window was previously enabled, FALSE if it was

not.

If the enable state is changed, a `WM_ENABLE` message is sent before `WinEnableWindow()` returns. This message is sent only if a state change occurs.

If the window is disabled, and it or one of its children is the focus window, then `WinSetFocus()` is called to set the focus window to `NULL`. However, the focus window may be set to a disabled window after the window is disabled.

When a window is disabled, its children are implicitly disabled, although they are not sent the `WM_ENABLE` message.

Notes     A disabled window does not respond to mouse input, but may still receive keyboard input if it is the focus window.

This function may be used to enable or disable ANY window, including the standard controls such as pushbuttons, scrollbars, etc.

Typically, a window changes its appearance when disabled. For example, a disabled pushbutton is displayed with halftone text.

---

## WinIsWindowEnabled

---

### Format

```
BOOL WinIsWindowEnabled(hwnd)
HWND hwnd;
```

### Description

Returns `TRUE` if `hwnd` is enabled, `FALSE` if it is disabled. A window is enabled if its `WS_DISABLE` window style bit is clear and disabled if the bit is set.

---

## WM\_ENABLE

---

### Format

```
WM_ENABLE
lParam1:    BOOL fEnable
lParam2:    OL
Returns:    OL
```

**Description**

The WM\_ENABLE message is sent from WinEnableWindow() if a window's enable state is changing. If fEnable is TRUE, a previously disabled window has been enabled. If fEnable is FALSE, a previously enabled window has been disabled.

This message is always sent before WinEnableWindow() returns, but AFTER the window enable state (WS\_DISABLE window style bit) has been changed.

This message is sent only if a state change has occurred.

#### *4.1.1.12.6 Showing or Hiding a Window*

Normally when a window is hidden, the windows underneath must redraw themselves clipped to the area being uncovered. However, if the CS\_SAVEBITS style is specified, the screen image underneath the window is saved when the window is made visible. When the window is hidden or destroyed, this window image is simply replaced onto the screen, avoiding any window repainting. If an application draws into a window underneath a CS\_SAVEBITS window, or if the window is sized or moved, the area of the saved image that is invalidated is redrawn when the window is hidden.

Because saved screen images can take up quite a bit of memory for large windows, the CS\_SAVEBITS class style should be used carefully. It is generally used for transient windows such as menus and dialog boxes, not for main application windows.

---

### **WinShowWindow**

---

**Format**

```
BOOL WinShowWindow(HWND hwnd, BOOL fShow)
```

```
HWND hwnd;  
BOOL fShow;
```

**Description**

This function is used to show or hide a window. If fShow is TRUE, the window is shown; if fShow is FALSE, the window is hidden. If the window is shown, it is made visible on the screen, and subsequent drawing in the window will be visible. If the window is hidden, the window is removed from the screen and all subsequent drawing in the window will not be visible.

The visible state is changed by setting or clearing

the `WS_VISIBLE` window style bit. The `WM_SHOW` message is sent AFTER the `WS_VISIBLE` bit is changed. `WinShowWindow()` returns `TRUE` if the `WS_VISIBLE` bit was previously set, `FALSE` if it was clear.

If the visible state of the window is changed, then the `WM_SHOW` message is sent before `WinShowWindow()` returns.

The `WM_SHOW` message is sent to the specified window before this function returns, but only if a state change has occurred. The message is sent AFTER the state change occurs.

---

## WinEnableWindowUpdate

---

### Format

```
BOOL WinEnableWindowUpdate (hwnd, fShow)
HWND hwnd;
BOOL fShow;
```

### Description

This function is similar to `WinShowWindow()`, except that no drawing is performed. If `fShow` is `FALSE`, the `WS_VISIBLE` window style bit is cleared, without removing the image of the window from the screen. Any subsequent drawing in that window will not be visible.

If `fShow` is `TRUE`, then the `WS_VISIBLE` bit is set, without actually causing the window to be redrawn. Subsequent drawing in the window will be visible, however.

`WinEnableWindowUpdate()` returns `TRUE` if the `WS_VISIBLE` bit was previously set, `FALSE` if it was clear.

The `WM_SHOW` message is sent to the specified window before this function returns, but only if a state change has occurred. The message is sent AFTER the state change occurs.

### Notes

This function is usually used to disable drawing before making a series of changes to a window to prevent needless drawing. To show a window and ensure that it is redrawn after calling `WinEnableWindowUpdate()` with `fShow == FALSE`, use `WinShowWindow()` with `fShow == TRUE`.

---

**WinIsWindowVisible**

---

**Format**

```
BOOL WinIsWindowVisible (hwnd)
HWND hwnd;
```

**Description**

Returns TRUE if the specified window and its parents are visible (i.e., have their WS\_VISIBLE window style bits set), FALSE otherwise.

This function does NOT simply return the state of the WS\_VISIBLE style bit of the specified window; the state of all of the parents of hwnd are tested as well.

**Notes**

WinIsWindowVisible() may return TRUE even if the window is completely obscured by other windows.

---

**WM\_SHOW**

---

**Format**

```
WM_SHOW
lParam1: BOOL fShow
lParam2: OL
Returns: OL
```

**Description**

This message is sent from WinShowWindow() after a visible window is hidden, or a hidden window is shown. If fShow is TRUE, a previously hidden window is being shown; if fShow is FALSE, a previously visible window is being hidden.

**Notes**

In this context, "Shown" or "Hidden" refers to the state of the WS\_VISIBLE style bit. This message is NOT sent when a window is obscured by other windows above it.

---

*4.1.1.12.7 Window Data Routines*

---

**WinQueryWindowText**

---

**Format**

```
INT WinQueryWindowText (hwnd, lpszBuffer, cchBufferMax)
HWND hwnd;
```

```
LPSTR lpszBuffer;
INT cchBufferMax;
```

Description

This function copies the null terminated window text of the specified window into \*lpszBuffer, returning the length of the string not including the null termination character. If the window's text is longer than cchBufferMax, it is truncated at (cchBufferMax - 1) characters; the string is always null terminated.

Notes

This function simply sends a WM\_QUERYWINDOWPARAMS message to the specified window.

If hwnd is a window of another process, then lpszBuffer must point to memory that is shared by both processes or a memory fault may occur.

If this function is called with a frame window handle, then the text of the title bar frame control is returned.

The WM\_QUERYWINDOWPARAMS message may be sent to determine the size of the string.

---

## WinQueryWindowTextLength

---

Format

```
INT WinQueryWindowTextLength(hwnd)
HWND hwnd;
```

Description

This function returns the length of the window text of the specified window, not including the zero-termination character. It sends a WM\_QUERYWINDOWPARAMS message to the window to obtain the information.

---

## WinSetWindowText

---

Format

```
BOOL WinSetWindowText(hwnd, lpszText)
HWND hwnd;
LPSTR lpszText;
```

Description

This function sets the text of the specified window to the string \*lpszText. lpszText is a far pointer

to a zero-terminated string. Returns TRUE if successful, FALSE otherwise.

**Notes**

This function simply sends a WM\_SETWINDOWPARAMS message to the specified window.

If hwnd is a window of another process, then lpzText must point to memory that is shared by both processes or a memory fault may occur.

If WinSetWindowText() is called with a frame window handle, the text of the title bar frame control is changed. See "Window Frames" for more information.

---

**WinSetWindowParams**

---

**Format**

```
BOOL fSuccess = WinSetWindowParams(lpszText,  
                                   lpCtlData, lpPresParams,  
                                   rgfStatus);  
  
LPSZ lpszText;  
UCHAR FAR *lpCtlData;  
UCHAR FAR *lpPresParams;  
UINT rgfStatus;
```

**Description**

This function sends a WM\_SETWINDOWPARAMS message. The parameters correspond to the parameters of the message. Returns TRUE if successful, FALSE otherwise.

---

**WinQueryWindowParams**

---

**Format**

```
INT cchText = WinGetWindowParams(lpszText,  
                                 cchTextMax, lpCtlData, lpPresParams,  
                                 rgfStatus);  
  
LPSZ lpszText;  
INT cchTextMax;  
UCHAR FAR *lpCtlData;  
UCHAR FAR *lpPresParams;  
UINT rgfStatus;
```

**Description**

This function sends the WM\_QUERYWINDOWPARAMS message. The parameters correspond to the parameters of the

message. The function returns the length of the window text if `WPM_TEXT` is specified in `rgfStatus`, otherwise 0 is returned.

---

### WinQueryDesktopWindow

---

#### Format

```
HWND hwndDesktop = WinQueryDesktopWindow (hab, hdc );  
HAB hab;  
HDC hdc;
```

#### Description

This function returns the desktop window handle for the device associated with `hdc`, or the default desktop window (screen) if `hdc` is `NULL`.

The call will fail if it is issued to a device which does not support windowing. This is any device which is not the screen device.

#### Notes

Many of the calls that require a desktop window handle will accept `NULL` instead of the desktop window handle. For example, `WinCreateWindow` will accept `NULL` for `hwndParent` in order to create a top level window, which is a child of the desktop window.

#### 4.1.1.12.8 Window data messages

```
typedef struct {  
    UINT cchText;  
    UINT cchTextMax;  
    LPSZ lpszText;  
    UCHAR FAR *lpCtlData;  
    UCHAR FAR *lpPresParams;  
} WNDPARAMS;
```

`lpszText` is a pointer to the control text, or `NULL`.

`lpCtlData` is a pointer to the control data, or `NULL`.

`lpPresParams` is a null pointer.

`cchText` is used only with the `WM_QUERYWINDOWPARAMS` message: the length of the text (excluding the zero termination character) is returned in this field. If `lpszText` is not `NULL`, this field contains the number of characters copied into `*lpszText`, not including the zero termination character. The maximum length in this case is `cchTextMax - 1`.



cchTextMax is used only with WM\_QUERYWINDOWPARAMS: it contains the size in characters of the text buffer pointed to by lpszText. This field is ignored by WM\_SETWINDOWPARAMS.

*rgfStatus values*

WPM\_TEXT       Set/Query control text  
WPM\_CTLDATA    Set/Query control data  
WPM\_PRESPARAMS Set/Query presentation parameters

---

## WM\_SETWINDOWPARAMS

---

Format

```
WM_SETWINDOWPARAMS
lParam1:       CTLPARAMS FAR *lpWndParams;
lParam2:       ULONG rgfStatus;
Returns:       BOOL fSuccess;
```

Description

This message is used to set the window data for a window. lParam1 is a far pointer to a wndparams structure defined above, and lParam2 contains the WPM status flags above.

Notes    If this message is sent to a window of another process, then \*lpWndParams, and the three data areas pointed to by \*lpWndParams, must all be in memory shared by both processes.

---

## WM\_QUERYWINDOWPARAMS

---

Format

```
WM_QUERYWINDOWPARAMS
lParam1:       CTLPARAMS FAR *lpWndParams;
lParam2:       ULONG rgfStatus;
Returns:       BOOL fSuccess;
```

Description

This message is used to query the window data for a window. lParam1 is a far pointer to a wndparams structure defined above, and lParam2 contains the WPM status flags above.

Notes    If this message is sent to a window of another process, then \*lpWndParams, and the three data areas pointed to by \*lpWndParams, must all be in memory shared by both processes.

#### 4.1.1.12.9 Window Information Functions

---

##### WinWindowFromID

---

###### Format

```
HWND WinWindowFromID(HWND hwndParent, ID id)
HWND hwndParent;
UINT id;
```

###### Description

This function returns the child window handle of `hwndParent` that has the specified window ID.

###### Notes

To obtain the window handle of a particular dialog item, `WinWindowFromID()` may be called with the dialog box window handle and the dialog item ID specified in the dialog template. See the "Dialog Manager" for more information.

---

##### WinMultWindowFromIDs

---

###### Format

```
INT WinMultWindowFromIDs(HWND hwndParent, LPARRAY,
                          ID idFirst, ID idLast)
HWND hwndParent;
HWND FAR *lprghwnd;
UINT idFirst, idLast;
```

###### Description

This function is used to quickly fill an array of window handles that have window ID values between `idFirst` and `idLast`, inclusive. The window handles are returned in `*lprghwnd`. `lprghwnd` is assumed to have  $(idLast - idFirst + 1)$  elements in the array. The windows are stored in the array indexed by their window ID: a window is stored in `lprghwnd[id - idFirst]`, where `id` is the window ID of the window. If no window exists with an ID in the range, that corresponding element in the array is `NULL`.

This function returns the number of window handles returned in the `lprghwnd` array. Returns 0 if no window handles were returned.

###### Notes

This function may be used to enumerate all items in a dialog group with contiguous ID values, or to enumerate all of the frame controls. This function is faster than multiple calls to `WinWindowFromID()`.

---

**WinQueryWindow**

---

**Format**

```
HWND WinQueryWindow(hwnd, code, fLock)
HWND hwnd;
INT code;
BOOL fLock;
```

**Description**

WinQueryWindow() returns a window handle associated with the specified window, depending on the code parameter. The available values of the code parameter and their effect is shown in the table below. If fLock is TRUE, then the window is returned locked, and the caller is responsible for unlocking it. If fLock is FALSE, the window is returned unlocked. See "Window Locking"

If WinQueryWindow() is used to enumerate windows of other threads, it is not guaranteed that all windows will be enumerated, because the Z ordering of the windows might change during the enumeration. Use WinEnumWindow() instead: see "Window Enumeration".

Available WinQueryWindow() codes:

---

**WinQueryWindow Codes**  
**Code**

QW_NEXT	Next window (window below)
QW_PREV	Previous window (window above)
QW_TOP	Top most child window
QW_BOTTOM	Bottom most child window
QW_OWNER	Owner of window
QW_PARENT	Parent of window. Returns HWND_OBJECT if object window.

---

**WinIsWindow**

---

Format

```
BOOL WinIsWindow(hab, hwnd)
HWND hwnd;
HAB hab;
```

Description

This function returns TRUE if hwnd is a valid window handle, FALSE otherwise.

---

WinIsChild

---

Format

```
BOOL WinIsChild(hab, hwndParent, hwnd)
HWND hwndParent;
HWND hwnd;
HAB hab;
```

Description

Returns TRUE if hwnd is a descendant of hwndParent, FALSE otherwise. If hwndParent is NULL, this function returns TRUE if hwnd is a top level window.

---

WinQueryWindowProcess

---

Format

```
UINT WinQueryWindowProcess(hwnd, lptid)
HWND hwnd;
UINT FAR *lptid;
```

Description

This function is used to obtain the process ID and thread ID associated with a window. Returns the process ID, or NULL if unsuccessful. The thread ID is returned in \*lptid. If lptid is NULL, the argument is ignored and the thread ID is not returned.

*4.1.1.12.10 Window Size & Position Related Functions*

---

WinQueryWindowRect

---

Format

```
void WinQueryWindowRect(hwnd, lprc)
HWND hwnd;
```

LPRECT lprc;

#### Description

This function returns the window rectangle of the specified window in \*lprc. The rectangle is returned in window coordinates relative to the bottom left corner of the parent of hwnd, or the screen if hwnd is a top level window.

---

### WinSetParent

---

#### Format

```
HWND WinSetParent(hwnd, hwndNewParent, fRedraw);  
HWND hwnd;  
HWND hwndNewParent;  
BOOL fRedraw;
```

#### Description

This function changes the parent window of hwnd to hwndNewParent, returning its previous parent handle.

If hwnd is visible and it is not an object window, the appropriate redrawing is performed.

If hwndNewParent is HWND\_OBJECT, hwnd becomes an "object window". Object windows can be converted to normal windows again on the screen by setting their parent to another normal window.

If hwndNewParent is NULL, hwnd becomes a top level window.

If hwnd is visible and fRedraw is TRUE, any necessary redrawing in both the old parent window and the new parent window is performed. If fRedraw is FALSE, no redrawing is performed. This is useful at times when the window will be later redrawn anyway, such as during WM\_SIZE message processing. The window is essentially first hidden in the old parent window, and shown again in the new parent window. The messages sent are the same as with WinShowWindow().

hwndNewParent may not be a child of hwnd.

#### Notes

This function returns an unlocked window handle. The WinQueryWindow() function can be used to get a window's parent.

---

**WinSetOwner**

---

**Format**

```
HWND WinSetOwner (hwnd, hwndNewOwner)
HWND hwnd;
HWND hwndNewOwner;
```

**Description**

This function changes the owner window of `hwnd` to `hwndNewOwner`, returning its previous owner window. `hwndNewOwner` may be `NULL` to disown a window. If the window was not previously owned, `NULL` is returned.

**Notes**

This function returns an unlocked window owner. The `WinQueryWindow()` function can be used to get a window's owner.

```
typedef struct {
    HWND hwnd;
    HWND hwndInsertBehind;
    INT x, y;
    INT cx, cy;
    UINT rgfSwp;
} SWP;
```

---

**WinSetWindowPos**

---

**Format**

```
BOOL WinSetWindowPos (hwnd, hwndInsertBehind,
                      x, y, cx, cy, rgfSwp)
HWND hwnd;
HWND hwndInsertBehind;
INT x, y, cx, cy;
UINT rgfSwp;
```

**Description**

`WinSetWindowPos()` is a general window positioning function, used to change position, size, and Z ordering of any window. Returns `TRUE` if successful, `FALSE` otherwise.

`hwnd` is the window to be changed. It is positioned `x` and `y` units from the bottom left corner of its parent, with the size specified by `cx` and `cy`. The window is inserted behind `hwndInsertBehind`.

If `hwndInsertBehind` is `HWND_TOP`, then `hwnd` is brought to the top, relative to its siblings. If `hwnd` is a top level window, it is activated if it is brought

to the top. If the active window is moved behind another window, the new top window is activated.

If `hwndInsertBehind` is `HWND_BOTTOM`, then `hwnd` is placed at the bottom relative to its siblings.

Not all of these parameters must be specified. The `rgfSwp` parameter controls the interpretation of the parameters, and the behavior of `WinSetWindowPos()`:

---

#### WinSetWindowPos Flags

Flag

##### SWP\_SIZE

Change the size of the window. Normally, the size is not changed.

##### SWP\_MOVE

Move the window. Normally, the window is not moved.

##### SWP\_ZORDER

Change the window Z ordering. Normally, the `hwndInsertBehind` parameter is ignored.

##### SWP\_NOREDRAW

Don't redraw changes. Normally, changes made are redrawn.

##### SWP\_ACTIVATE

Activate the window if the window is brought to the top, and deactivate the window if it is moved from the top to behind another window. The new top-most window is activated. Normally, the active window is not changed.

##### SWP\_ADJUST

Send a `WM_ADJUSTWINDOWRECT` message before moving or sizing window.

#### Notes

Messages may be received from other processes or threads.

If a window is made smaller and it has the `CS_SAVEBITS` style, the saved screen image is used to redraw the area uncovered when the window size changes, if those bits are still valid. See "Showing and Hiding Windows".

If the `CS_SIZEREDRAW` style is present, the entire window area is assumed invalid if sized.

Otherwise `WM_CALCVALIDRECTS` is sent to the window to inform the window manager which bits may be possible to preserve.

Messages sent from `WinSetWindowPos()` and `WinSetMultWindowPos()` have specific orderings within the window positioning process. The process begins with redundancy checks and precalculations on every window for each requested operation. For example, if `SWP_SHOW` is present but the window is already visible, `SWP_SHOW` is turned off. If `SWP_SIZE` is present and the new size is equal to the old size, `SWP_SIZE` is turned off, etc. If the operations will create new results, the information is calculated and store away. For example, if sizing or moving, the new window rectangle is stored away for later. It is at this point that the `WM_ADJUSTWINDOWRECT` message is sent to any window that is sizing or moving. It is also at this point that the `WM_CALCVALIDRECTS` message is sent to any window that is sizing and doesn't have the `CS_SIZEREDRAW` class style.

Once all the new window state is calculated, the window management process begins. Window areas that can be preserved are moved from the old to the new positions, window areas that are invalidated by these operations are calculated and distributed as update regions, etc. Once this is finished, and before any sync paint windows are repainted, the `WM_SIZE` message is sent to any windows that have changed size.

at Next, all the sync paint windows that can be repainted are repainted, and the entire process in complete.

If a sync paint parent window has a size sensitive area displayed that includes sync paint child windows, the parent will want to reposition those windows when it gets the `WM_SIZE` message. Their invalid regions will get added to the parent's, resulting in one update after the parent's `WM_SIZE` message, rather than many independent and later duplicated updates.

Messages sent from `WinSetWindowPos()`:

---

Message	When Sent
---------	-----------



**WM\_CALCVALIDRECTS**

This message is sent to determine the area of a window that may be possible to preserve as the window is sized.

See "Window Painting" for more information on the WM\_CALCVALIDRECTS message.

**WM\_SYNCPAINT**

This message is sent when any part of a window with the WS\_SYNCPAINT class style becomes uncovered or otherwise requires repainting. May be sent to windows of other processes or threads.

**WM\_SIZE**

This message is sent if the size of the specified window has changed. Sent AFTER the window's size has changed.

**WM\_ACTIVATE**

If a top level window is brought to the top and SWP\_NOACTIVATE is not specified, the window is activated. In this case, WinSetActiveWindow() will send a WM\_ACTIVATE message, and possibly others. See WinSetActiveWindow() for more information.

**WM\_ADJUSTWINDOW ECT**

This message is sent if SWP\_ADJUST is specified. lParam1 points to an SWP structure which has been filled in by SetWindowPos() with the proposed move/size data. The window may adjust this new position by changing the contents of the SWP structure. It can change the x/y fields to adjust its new position; it can change the cx/cy fields to adjust its new size, or it change the hwndInsertBehind field to adjust its new z-order.

---

**WinSetMultWindowPos**

---

**Format**

```
BOOL WinSetMultWindowPos(hab, lpSwp, cSwp)
HAB hab;
SWP FAR *lpSwp;
INT cSwp;
```

### Description

This function is similar to `WinSetWindowPos()`, except that it can be used to reposition more than one window at a time. `lpSwp` points to an array of `SWP` structures, and `cSwp` is the count of `SWP` structures in that array. A single call to `SetMultWindowPos()` is much faster and causes less screen updating than multiple calls to `WinSetWindowPos()`.

All the windows concerned must have the same parent.

The fields of the `SWP` structure correspond to the parameters of `SetWindowPos()`.

Returns `TRUE` if any error occurred, `FALSE` otherwise.

---

## WM\_SIZE

---

### Format

```
WM_SIZE
LOUINT(lParam1): INT cxNew
HIUINT(lParam1): INT cyNew
LOUINT(lParam2): INT cxOld;
HIUINT(lParam2): INT cyOld;
Returns:         OL
```

### Description

This message is sent from `WinSetWindowPos()` or `WinSetMultWindowPos()` when a window's size is initialized or changed. `cxOld` and `cyOld` are the PREVIOUS horizontal and vertical dimensions of the window. The `LOUINT` of `lParam1` contains the new width of the window, the `HIUINT` of `lParam1` contains the new height of the window.

This message is NOT sent by `WinCreateWindow()` when the window is created. Any size related processing must be performed during the `WM_CREATE` message processing.

This message is sent after the window has been actually sized, but before any repainting has been performed. Any resizing or repositioning of child windows, etc., that may be necessary as a result of the size change is usually performed during the processing of this message. It is generally unwise to do any output to the window during the processing of the `WM_SIZE` message, because the area drawn into may be drawn a second time after

the WM\_SIZE processing is complete by WinSetWindowPos().

The processing of this message for a window which is displaying an Advanced Vio presentation space must be carried out by WinDefAVioWindowProc. See the section, “WM\_SIZE Message Processing”, for details.

---

## WM\_MOVE

---

### Format

WM\_MOVE  
lParam1: OL  
lParam2: OL  
Returns: OL

### Description

The window manager sends a WM\_MOVE message when a window with WS\_MOVENOTIFY style changes its absolute position (ie. relative to the origin of the screen). It is sent from WinSetWindowPos(), WinSetMultWindowPos(), and WinScrollWindow().

The window's new position is obtained by calling WinGetWindowRect(), and can make those rectangle coordinates relative to any window by calling WinMapWindowPoints().

There are several cases where windows want to know if they've been moved. This includes the cases when the window doesn't change position relative to its parent but does change position relative to the screen (its absolute position).

An example is menus. When a top level menu control (child of the frame window) moves its absolute position as a result of the frame window being moved, the top level menu control wants to move any pull down menus along with its movement. The same goes for application / dialog box positional grouping. In some cases a dialog box might want to be moved as the main window is moved, to clear room for other applications.

---

## WM\_CALCVALIDRECTS

---

## Format

```
WM_CALCVALIDRECTS  
lParam1: RECT FAR *lprcWindowOld;  
lParam2: RECT FAR *lprcWindowNew;  
Returns: UINT Align;
```

## Description

This message is sent from `WinSetWindowPos()` and `WinSetMultWindowPos()` to determine which areas of a window may be preserved if a window is sized, and which should be redisplayed. This message is NOT sent if this window has the `CS_SIZEREDRAW` style, indicating size sensitive window content that must be totally redrawn if sized (see `WinSetWindowPos()`).

`lParam1` points to a rectangle structure that contains the rectangle of the window before the move. `lParam2` points to the new window rectangle. The coordinates of the rectangles are parent window relative. This allows the application to determine if the window's position has changed as well as its size, which can aid alignment processing.

These rectangles may be modified by the window procedure to cause parts of the window to be redrawn and not preserved.

The window manager will attempt to preserve the screen image by copying the image described by the old rectangle into the image described by the new rectangle. In this way, an application may control the alignment of the preserved image as well, by changing the origin of the first rectangle.

If no change is made to either rectangle, the entire window area is preserved. If either rectangle is empty, the entire window area is completely redrawn by the operation. The message has a return value, `Align`, which can be used to instruct `WinSetWindowPos()` how to align valid window bits. This return value is made up from `CVR_` flags, as follows:

---

`CVR_ALIGNLEFT`

Align with the left edge of the window

`CVR_ALIGNBOTTOM`

Align with the bottom edge of the window

**CVR\_ALIGNTOP**

Align with the top edge of the window

**CVR\_ALIGNRIGHT**

Align with the right edge of the window

**CVR\_REDRAW**

The whole window is invalid

If 0 is returned, it is assumed the application has changed the rectangles pointed to by lParam1 and lParam2 itself.

If CVR\_REDRAW is set, the whole window is assumed invalid. Otherwise, the remaining flags may be OR'ed together to get differing kinds of alignment. For example:

(CVR\_ALIGNLEFT | CVR\_ALIGNTOP)

Align the valid window area with the top left of the window.

The WinDefWindowProc() default is to return CVR\_REDRAW if the window has the style CS\_SIZEREDRAW, and otherwise align the valid area with the window origin by returning (CVR\_ALIGNBOTTOM | CVR\_ALIGNLEFT).

In addition, any child windows intersecting the source rectangle pointed to by lParam1 of the WM\_CALCVALIDRECTS message will also be offset with the aligned window area.

This functionality can be used to optimize window updating when the window is resized. For example, if the application returns that the window is to be aligned with the top left corner, and the top border is sized, the window's screen data will move with the top border.

In all cases, the rectangles are intersected with the area of the screen that is actually visible and the valid area of the window. That is, only the window area that contains window information is copied.

For example, consider an application that has two scroll bars, which are children of the client window. When the window is resized, the scroll bars must be completely redrawn. By returning rectangles that exclude the scroll bars, the area of the scroll bars is completely redrawn, thereby preserving only that part of the screen that is worth preserving.

#### 4.1.1.12.11 Window Subclassing Functions

---

##### WinSubclassWindow

---

###### Format

```
FARPROC WinSubclassWindow(hwnd, lpfnNewWindowProc)
HWND hwnd;
FARPROC lpfnNewWindowProc;
```

###### Description

This function is used to subclass the specified window. The window procedure of the specified window is replaced with lpfnNewWindowProc, which is a far procedure address of a window procedure. This function returns the specified window's previous window proc address.

###### Note

When subclassing, the new procedure address should call the old procedure address in place of calling WinDefWindowProc().

To reverse the effect of subclassing, simply call WinSubclassWindow() with the previous window procedure address.

This routine does not allow the application to subclass a window associated with another process.

If this function fails, zero (LONG) is returned.

```
typedef struct {
    ULONG styleClass;
    FARPROC lpfnWindowProc;
    INT cbWndExtra;
    UINT idModule;
} CLASSINFO;
```

---

##### WinQueryClassInfo

---

###### Format

```
BOOL WinQueryClassInfo(hab, lpstrClassName,
                        lpClassInfo)
HAB hab;
LPSTR lpstrClassName;
CLASSINFO FAR *lpClassInfo;
```

###### Description

This function is used to obtain information about the specified window class. Returns TRUE if the class named by \*lpstrClassName exists, FALSE

otherwise. If the class exists, information about the class is returned in \*lpClassInfo.

Notes     This function is used to create subclasses of a given class.

#### *4.1.1.12.12 Window Enumeration Functions*

The three functions WinBeginEnumWindows(), WinEnumWindow(), and WinEndEnumWindows() are used to enumerate windows that might be owned by different processes or threads. Any one application can not make any assumptions about a window owned by another process or thread unless that window is locked. Instead of locking all the windows in the list to be enumerated, Presentation Manager makes a snapshot copy of the list of window handles to be enumerated, which is represented by an 'enumeration' handle. The application uses this handle for subsequent enumeration calls to get at the windows in this list. There is a final call which ends the enumeration and frees the enumeration handle. See "Window Locking".

Below is an example of how these functions are used to enumerate windows:

```
/*
 * Enumerate all top level windows
 */
hEnum = WinBeginEnumWindows (NULL);

/*
 * For each window handle returned from WinEnumWindow(), call our
 * internal function DoSomething(), until we've enumerated the
 * entire list.
 */
while ((hwnd = WinEnumWindow(hEnum)) != NULL) {
    DoSomething(hwnd);
    /*
     * Be sure to unlock the window when finished with it.
     */
    UnlockWindow(hwnd);
}

/*
 * Finish the enumeration
 */
WinEndEnumWindows (hEnum);
```

Window enumeration using these functions may be nested; however, a call to WinEndEnumWindows() must be made with the handle returned from its corresponding call to WinBeginEnumWindows().

---

## WinBeginEnumWindows

---

### Format

```
HENUM WinBeginEnumWindows(hab, hwndParent)
HWND hwndParent;
HAB hab;
```

### Description

This function begins the enumeration of all the children of `hwndParent`, returning a handle to be used when enumerating these windows. Regardless of any subsequent changes to the window Z ordering, the windows are enumerated in the Z ordering that existed at the time `WinBeginEnumWindows` was called. This ensures that all of the child windows of `hwndParent` are enumerated.

If `hwndParent` is `NULL`, all top level windows are enumerated.

Windows may be destroyed after `WinBeginEnumWindows()` is called.

See the example above.

---

## WinEnumWindow

---

### Format

```
HWND WinEnumWindow(hEnum)
HENUM hEnum;
```

### Description

This function returns the next window to be enumerated. `hEnum` is a handle returned by `WinBeginEnumWindows()`. Each call to this function returns the next window to be enumerated. `NULL` indicates that all windows have been enumerated.

`WinEnumWindow()` always returns a locked window handle, which must be unlocked by the caller.

See example above.

---

## WinEndEnumWindows

---

### Format

```
BOOL WinEndEnumWindows(hEnum)
HENUM hEnum;
```



**Description**

This function ends the enumeration of the windows. hEnum is the handle returned by a previous call to WinBeginEnumWindows().

See example above.

*4.1.1.12.13 Window Hit Testing*

"Hit Testing" refers to the process of determining what object a mouse point is in. Window hit testing is the process of determining which window a given mouse point is within.

---

**WinWindowFromPoint**

---

**Format**

```
HWND WinWindowFromPoint(hab, hwndParent,
                        lppt, fEnumChildren)
HAB hab;
HWND hwndParent;
POINT FAR *lppt;
BOOL fEnumChildren;
```

**Description**

Returns the window handle underneath the specified point, by checking to see if \*lppt is within the window rectangles of the children of hwndParent. Returns hwndParent if the point is not inside any of the children of hwndParent, but is inside hwndParent. Returns NULL if the point is outside hwndParent.

\*lppt must be specified in window coordinates, relative to hwndParent.

If hwndParent is NULL, then all top level windows are enumerated. In this case, \*lppt must be relative to the top left corner of the screen.

If fEnumChildren is TRUE, all descendants of hwndParent are hit tested; otherwise, only immediate children of hwndParent are hit tested.

#### 4.1.1.12.14 *Coordinate Mapping*

---

##### WinMapWindowPoints

---

###### Format

```
void WinMapWindowPoints(hab, hwndFrom, hwndTo,
                        lprgpt, cpt)
HAB hab;
HWND hwndFrom;
HWND hwndTo;
POINT FAR *lprgpt;
INT cpt;
```

###### Description

Maps hwndFrom relative window coordinate points to hwndTo relative window coordinate points. lprgpt is a far pointer to an array of POINT structures to map, and cpt is the number of structures in the array.

If hwndFrom is NULL, the points in the array are assumed to be in screen coordinates.

If hwndTo is NULL, the points in the array will be mapped to screen coordinates.

lprgpt may be used to point to a RECT structure; in this case cpt must be 2.

###### Examples

```
/* map from window to screen coordinates */
WinMapWindowPoints(hwndFrom, NULL, lppt, cpt);

/* map from screen to window coordinates */
WinMapWindowPoints(NULL, hwndTo, lppt, cpt);

/* map from child to parent window coordinates */
WinMapWindowPoints(hwndChild,
                    WinQueryWindow(hwndChild, QW_PARENT, FALSE),
                    lppt, cpt);
```

#### 4.1.1.12.15 *Accessing Window Words*

For window classes registered by applications, it is possible to reserve extra memory for each window that can be used by the window procedure to store any extra information that may be required for the window class.

When a window class is registered, the number of extra bytes to reserve is specified. Every window instance of that class will have these bytes allocated and initialized to 0, which can be read and written by the application or window procedure.

These functions are used to access this reserved memory. In addition, there are special values that can be used to access certain internal window fields, as shown in the table below. `QWL_` constants are for use with `WinQueryWindowUInt()` and `WinSetWindowUInt()`, and `QWL_` constants are for use with `WinQueryWindowULong()` and `WinSetWindowULong()`:

---

Standard `WinQueryWindowUInt` Indexes

<code>QWL_HMQ</code>	Message queue handle
<code>QWL_ID</code>	Window ID (ID passed to <code>WinCreateWindow()</code> )
<code>QWL_STYLE</code>	Window style
<code>QWL_WNDPROC</code>	Window Procedure address

`QWL_USER`

The following preregistered window classes have a window `ULONG` at `WinSet/QueryWindowULong()` offset `QWL_USER` available for application use:

- `WC_DIALOG`
- `WC_FRAME`
- `WC_LISTBOX`
- `WC_BUTTON`
- `WC_STATIC`
- `WC_EDIT`
- `WC_SCROLLBAR`
- `WC_MENU`

This is useful for placing application-specific data in controls.

*Note:* `QWL_` constants should NEVER be used with `WinQueryWindowULong()` or `WinSetWindowULong()`, and `QWL_` constants should NEVER be used with `WinQueryWindowUInt()` or `WinSetWindowUInt()`.

---

`WinQueryWindowUInt`

---

Format

```
UINT WinQueryWindowUInt(hwnd, ib)
HWND hwnd;
```

INT ib;

---

### WinSetWindowUInt

---

#### Format

```
UINT WinSetWindowUInt (hwnd, ib, w)
HWND hwnd;
INT ib;
UINT w;
```

#### Description

These functions are used to read or write 16 bits of information in the reserved window word memory associated with the specified window, at index ib. ib may also be one of the QWL\_ values shown in the table above.

ib is a zero-based index into the window words. Only values between 0 and cbWndExtra may be used, or any of the QWL\_ values shown in the table above.

WinQueryWindowUInt() returns the unsigned 2 byte integer at the index specified by ib. WinSetWindowUInt() stores the unsigned 2 byte integer w at the index specified by ib, and returns the previous contents of the UInt.

These functions are used to read the window words associated with the specified window, using the index ib. ib may also be one of the QWL\_ values in the table above.

**Notes** The QWL\_ constants may not be used with these functions.

---

### WinQueryWindowULong

---

#### Format

```
LONG WinQueryWindowULong (hwnd, ib)
HWND hwnd;
INT ib;
```

---

### WinSetWindowULong

---

#### Format

```
LONG WinSetWindowULong (hwnd, ib, lData)
HWND hwnd;
```

```
INT ib;  
ULONG lData;
```

**Description**

These functions are used to read or write 32 bits of information in the reserved window word memory associated with the specified window, at index `ib`. `ib` may also be one of the `QWL_` values shown in the table above.

`ib` is a zero-based index into the window words. Only values between 0 and `cbWndExtra` may be used, or any of the `QWL_` values shown in the table above.

`WinQueryWindowULong()` returns the long at the index specified by `ib`. `WinSetWindowULong()` stores the long `lData` at the index specified by `ib`, and returns the previous contents of the long.

**Notes**

The `QWL_` constants may not be used with these functions.

#### *4.1.1.12.16 Active Window Routines*

---

**WinSetActiveWindow**

---

**Format**

```
BOOL WinSetActiveWindow(hab, hwnd)  
HWND hwnd;  
HAB hab;
```

**Description**

This function is used to set the active window to the specified window. Returns `TRUE` if successful, `FALSE` otherwise.

**Notes**

This function sets the keyboard focus to `NULL`, and does not directly set the focus. Typically, the focus is set during the processing of the `WM_ACTIVATE` message by the window being activated.

`WinSetActiveWindow()` should not be called unless it is directly or indirectly a result of user input.

`WinSetActiveWindow()` succeeds only in the following conditions:

1. It is being called in the context of the thread that is currently associated with the active application.

2. It is being called in the context of a thread that is currently processing a message from another application. See `WinInSendMessage()`.
3. It is being called when an application is being started.

Messages may be received from other processes or threads if either the current active window or the new active window is associated with another thread or process.

The following messages are sent by `SetActiveWindow`. No messages are sent if `hwnd` is the same as the current active window:

---

Message

When Sent

#### WM\_ACTIVATE

This message is sent first to the window being deactivated with `LOUINT(lParam) == FALSE`. Then, this message is sent to the window being activated with `LOUINT(lParam) == TRUE` and `HUINT(lParam) == TRUE`.

Note that a `WM_SETFOCUS` message is sent to the window which is losing the focus (if any).

During the processing of a `WinSetActiveWindow()` call, if `WinGetActiveWindow` or `WinGetFocus()` are called, the old active and focus windows are returned until the new ones have been established. In other words, even though `WM_ACTIVATE(false)` messages may have been sent to the old windows, those old windows are considered to be active and have the focus (until the system establishes the new active and focus windows).

---

### WinGetActiveWindow

---

Format

```
HWND WinGetActiveWindow(hab, fLock)
HAB hab;
BOOL fLock;
```

Description

This function returns the current active window, or `NULL` if there is no active window. If `fLock` is

TRUE, then the window is returned locked, and the caller is responsible for unlocking it. If fLock is FALSE, the window is returned unlocked. See "Window Locking"

---

## WM\_ACTIVATE

---

### Format

```
WM_ACTIVATE
LOUINT(lparam1): BOOL fActive;
HIUINT(lParam1): BOOL fSetFocus;
lparam2       : HWND hwndActive;
Returns       : BOOL fProcessed;
```

### Description

This message is sent by WinSetActiveWindow(), WinSetFocus(), WinCreateWindow(), WinShowWindow(), WinSetWindowPos(), or WinSetMultWindowPos() when a window is activated or deactivated. If LOUINT(lParam1) is TRUE, the window is being activated; if FALSE, the window is being deactivated.

When LOUINT(lParam1) is FALSE, an application should save away the current focus window, for later restore when the window is reactivated. lParam2 has the window handle of the window being activated.

When LOUINT(lParam1) is TRUE and HIUINT(lParam1) is TRUE, the application should set the focus to the saved focus window. If HIUINT(lParam1) is FALSE, the message is being sent as a result of a WinSetFocus() call, therefore the application should NOT set the focus. lParam2 has the window handle of the window being deactivated.

The default WinDefWindowProc() behaviour is to simply set the focus to the window being activated if LOUINT(lParam1) and HIUINT(lParam1) are both TRUE. WinDefWindowProc() does nothing otherwise.

### Notes

WM\_ACTIVATE with LOUINT(lParam1) == FALSE is sent before WM\_ACTIVATE with LOUINT(lParam1) == TRUE. Any WM\_SETFOCUS messages with lparam2 == FALSE are sent BEFORE the deactivation message. WM\_SETFOCUS messages with lparam2 == TRUE are sent AFTER the activation

message.

If `WinSetFocus()` is called during `WM_ACTIVATE` or `WM_ACTIVATETHREAD` message processing, a `WM_SETFOCUS` message with `fFocus == FALSE` is *not*

Except in the case of the `WM_ACTIVATE` message with `fActive == TRUE`, an application processing `WM_SETFOCUS`, `WM_ACTIVATE`, or `WM_ACTIVATETHREAD` should not change the focus window or active window. If it does, focus and active window must be restored before the application returns from processing the message. For this reason, any dialog boxes or windows brought up during `WM_SETFOCUS`, `WM_ACTIVATE` or `WM_ACTIVATETHREAD` processing should be system modal.

---

## WM\_ACTIVATETHREAD

---

### Format

```
WM_ACTIVATETHREAD
lParam1:      BOOL fActive:
LOUINT(lParam2): UINT idProcess;
HIUINT(lParam2): UINT idThread;
Returns:      OL
```

### Description

This message is sent by the `WinSetActiveWindow()`, `WinSetFocus()`, `WinCreateWindow()`, `WinShowWindow()`, `WinSetWindowPos()`, or `WinSetMultWindowPos()` functions when a window is being activated or deactivated, and the process/thread that owns the window being activated is different to the process/thread that owns the current active window.

- If `lParam1 == TRUE`, the window is being activated, and `lparam2` identifies the process/thread associated with the window that was the previous active window, or `NULL` if there was no previous active window.
- If `lParam1 == False`, the window is being deactivated, and `lparam2` identifies the process/thread associated with the window that will become the new active window, or `NULL` if there will be no active window.



---

**WinIsThreadActive**

---

**Format**

```
BOOL WinIsThreadActive (hab)
HAB hab;
```

**Description**

This function returns TRUE if the active window is associated with the current thread.

*4.1.1.12.17 Window Flashing*

---

**WinFlashWindow**

---

**Format**

```
BOOL WinFlashWindow (hwnd, fFlash)
HWND hwnd;
BOOL fFlash;
```

**Description**

This function is used to start or stop window flashing. If fFlash is TRUE, window flashing is begin; if fFlash is FALSE it is stopped. Returns TRUE if successful, FALSE otherwise.

A window is normally flashed by inverting the state of the title bar. A beep is emitted for the first 5 flashes.

**Notes**

This function is used when it is necessary to bring up a dialog box or message box when the application is not the current application. The flashing window indicates that the user's attention is required -- when the user activates the window, the flashing stops and the message box or dialog box is brought up.

*4.1.1.12.18 System Modal Window Routines*

The "System Modal Window" is a special top level window that receives all mouse and keyboard input. Input may also be routed to one of its children. All other top level windows behave as if they are disabled; no interaction is possible.

If another window is explicitly set to the active window, the newly activated window is set to the system modal window. The previous system modal window may no longer be interacted with. If a system modal window is destroyed, the window activated as a result becomes the system modal window.

Non-system modal windows are not actually disabled with `WinEnableWindow()`; they are simply made non-interactive. No messages are sent, and the `WS_DISABLE` style bits are not changed.

---

### WinGetSysModalWindow

---

#### Format

```
HWND WinGetSysModalWindow(hab, fLock)
HAB hab;
BOOL fLock;
```

#### Description

This function returns the window handle of the current system modal window, `NULL` if there is none. If `fLock` is `TRUE`, then the window is returned locked, and the caller is responsible for unlocking it. If `fLock` is `FALSE`, the window is returned unlocked. See "Window Locking"

---

### WinSetSysModalWindow

---

#### Format

```
HWND WinSetSysModalWindow(hab, hwnd)
HWND hwnd;
HAB hab;
```

#### Description

This function sets the current system modal window to `hwnd`, and returns the previous system modal window handle. The window handle is returned unlocked.

If `hwnd` is `NULL` (i.e., there is no system modal window), other windows receive mouse input again as usual.

## 4.1.2 Window Drawing Management Architecture

### 4.1.2.1 The Window DC

Graphical output to a window is done to a Presentation Space, like any other graphics output. However, in order to draw in a window, the PS must be associated with a special kind of device context (DC) called a "Window DC". A window DC may be created for a window, and it is automatically destroyed when its associated window is destroyed.

As explained earlier in the section on "Window Management", not all of a window is necessarily visible on the screen; a window may be obscured by windows above, and clipped to its parent. The window DC associated with a window allows access only to the part of the screen that corresponds to the part of the window that is actually visible.

The area of a window that is actually visible is called the "visible region", or "visrgn" for short. This visrgn is part of a window DC, and is always maintained by the window manager as windows are rearranged. It cannot be changed by the application. All output to a PS attached to a window DC is clipped to the visrgn of the window DC. The origin of the PS is always the bottom left corner of the window rectangle, regardless of where it is located on the screen.

This visrgn clipping does not affect the normal GPI clipping. An application is free to change the PS clipping area as it wishes. Physically, however, output is clipped to the intersection of the PS clipping area and the window DC visrgn.

Getting ready to draw into a window requires three steps:

1. Create a window
2. Create a PS
3. Create a window DC for the window created above.
4. Associate the window DC with the PS. Now, any drawing to the PS will be drawn in the visible part of the window.

These steps are very similar to those required for drawing on any other graphical device.

If you want to use a micro-PS instead of a standard PS in order to save memory, here's what you have to do:

1. Create a window
2. Create a window DC for the window created above

3. Create a micro-PS from the window DC created above.

#### 4.1.2.2 Cached Micro-PS

There is another way that an application may obtain a micro-PS for use in drawing in a window, that does not require the creation of a window DC or PS.

The window manager maintains a cache of micro-PSs and window DCs that an application may use. To obtain a micro-PS, the function `WinGetPS()` is called with the window to be drawn in. The micro-PS obtained from the cache is associated with a window DC for the specified window.

In order to allow other applications access to the cache, the micro-PS must be released by calling `WinReleasePS()` when it is no longer required.

When a micro-PS is obtained from the cache, its state (colors, transforms, clipping state, etc) is the same when a PS is first created. Any state changes made to a cached PS are lost when it is released. The origin of the PS is always aligned with the window rectangle.

There are two key advantages to using a cached PS: space and speed.

A normal PS occupies about 1K of memory. A window DC occupies about 400 bytes. Creating a PS and an DC takes a fair amount of execution time, as well.

Cached PSs do not require any additional memory. Obtaining and releasing a PS from the cache is much faster than creating and destroying a PS and a window DC.

These considerations are especially important with some windows, such as dialog boxes, that may contain 20 or 30 child windows. If each of those windows had its own PS and window DC, some 30K to 45K of memory would be occupied by all the PS's and DC's.

#### 4.1.2.3 Application PS vs. Cache PS Considerations

Application PSs are appropriate when a window stays around for a long period, such as with main application windows. They should also be used if non-retained graphics are being used, or if any of the features of a PS not available with micro-PSs is desired.

Application PSs are faster, if windows are not being created and destroyed. They are also useful if there are lots of PS state changes required, or a normal (non-micro-) PS is required. Visrtn calculation takes place after each window management operation, though. They also take

memory.

Cached PSs are faster if windows are being created and destroyed often. Getting a cache entry the first time causes a region calculation; subsequent calls are cheap. Since state is lost when a cached PS is released, it is sometimes expensive to have to select state in every time a PS is obtained.

#### 4.1.2.4 Window Clipping Options

An application has some control over how windows are clipped. There are two window style bits and a class style bit that control window clipping. These styles control:

1. Whether a window excludes the area of its children
2. Whether a window excludes the area of its siblings above
3. Whether a window is clipped to the window rectangle, or to its parent's visrgn.

A window may be created with one or both of the following window or class styles to change the clipping behavior:

---

Window/Class Style	Behavior
--------------------	----------

<b>WS_CLIPCHILDREN</b>	
------------------------	--

	The area of a window's children is excluded from the window. If this style is not specified, the children are not excluded.
--	---

<b>WS_CLIPSIBLINGS</b>	
------------------------	--

	The area of a window's siblings above the window is excluded from the window. If this style is not specified, the siblings are not excluded.
--	--

<b>CS_PARENTCLIP</b>	
----------------------	--

	The parent's visrgn is used for clipping. The origin of the PS is still the bottom left corner of the window, but drawing may be done outside of the window rectangle. Output is clipped to the visible region of the parent. The children of the parent are not excluded, even if the parent has the WS_CLIPCHILDREN style. If this class style is not specified, the window is clipped to its window rectangle.
--	---

#### 4.1.2.5 Window Clipping Considerations

The key reason for allowing these clipping options is speed. It takes a certain time to perform the clipping region calculations that are required.

For cached PSs, these calculations are performed if there are no valid cache entries for the window. A cache entry is invalidated from the cache any time window rearrangement, hiding, or showing is done that might affect the visrgn of the window. When a cached micro-PS is returned by WinGetPS(), it is considered to be "in use" until it is released with WinReleasePS(). Cache entries that are not in use may also be reused for other windows as required.

For windows with permanent window DCs, these calculations are performed every time the window visible region changes as a result of any window rearrangement, hiding, or showing that affects the visrgn of the window.

---

Style	Recommendation
-------	----------------

WS_CLIPCHILDREN	
-----------------	--

	The WS_CLIPCHILDREN style should only be used if it is necessary to prevent a parent window from drawing on its children. If both the parent and the child are invalidated at once, the painting order will be top down; the parent will draw before the child draws. If the child is invalidated independently of the parent, the child will draw independently of the parent.
--	---

WS_CLIPSIBLINGS	
-----------------	--

	WS_CLIPSIBLINGS should generally be used when child windows physically overlap. Sometimes it may be avoided even if they do overlap: since sibling windows are always repainted in front to back order, the image on the screen may be deterministic.
--	---

CS_PARENTCLIP	
---------------	--

	The CS_PARENTCLIP style is normally used by non-overlapping windows with a common parent. All windows with CS_PARENTCLIP and having the same parent window will use the same visrgn, and therefore the same origin-modified PS for drawing. This avoids subsequent visrgn calculations for each window, a drawing speed gain. It makes no sense for a CS_PARENTCLIP window to have either the WS_CLIPCHILDREN or WS_CLIPSIBLINGS styles defined, as they defeat the purpose of the CS_PARENTCLIP style. Most of the standard controls are CS_PARENTCLIP windows, and none of them draw outside their window rectangle. Generally a CS_PARENTCLIP window should not draw outside its window rectangle.
--	---

#### 4.1.2.6 Application PS Example

Application initialization:

```
hwnd = WinCreateWindow(...);          /* Create a window      */
hps = GpiCreatePS(...);                /* Create a PS for graphics */

hdcWindow = WinOpenWindowDC(hwnd);     /* Create a DC for output to */
                                      /* window                    */

hdcPrinter = GpiCreateDC("Printer");    /* Create a DC for output to */
                                      /* printer                   */

GpiAssocDC(hps, hdcWindow);            /* Send output to screen    */
                                      /* window                   */
DrawDocument(hps);                    /* Draw the document        */
```

WM\_PAINT message processing:

```
WinBeginPaint(hwnd, hps, (LPRECT)&rcPaint);

DrawDocument(hps);                    /* Redraw the document      */

WinEndPaint(hps);
```

Printing:

```
GpiAssocDC(hps, hdcPrinter)           /* Send output to printer   */
DrawDocument(hps);                    /* Draw the document        */
```

#### 4.1.2.7 Cached-PS Example

```
hwnd = WinCreateWindow(...);          /* Create a window      */
hps = WinGetPS(hwnd);                  /* Get cache PS for drawing */

DrawDocument(hps);                    /* Draw the document      */
WinReleasePS(hps);                    /* Release the cache PS    */
```

WM\_PAINT message processing:

```
hps = WinBeginPaint(hwnd, NULL, (LPRECT)&rcPaint);

DrawDocument(hps);                    /* Redraw the document      */

WinEndPaint(hps);
```

#### 4.1.2.8 Window Repainting after Window Rearrangement

When windows are rearranged or shown and a new area of the window is made visible, it is the responsibility of the window procedure to redraw the image in the area of the window that has been uncovered. This update repainting may be done synchronously at the time the rearrangement takes place, or asynchronously some time after the rearrangement operation, when an application chooses to redraw the window. The `CS_SYNCPAINT` style bit controls whether a window is updated synchronously or asynchronously.

##### *4.1.2.8.1 Asynchronous Window Updating*

When some part of a window not having `CS_SYNCPAINT` style is made visible due to window rearrangement or showing of the window, the window is not painted right away. Instead, the area to be updated is saved in the window's "Update Region". The update region is the area of a window that will eventually require repainting. As subsequent areas are made visible due to additional window rearrangement, these areas are accumulated (added) to the window's update region.

Accumulating an area into the update region is called "Window Invalidation". Removing area from the update region is called "Window Validation". In to the invalidation and validation that occurs as a result of window rearrangement, An application can explicitly invalidate and validate areas of a window.

When an application calls `WinGetMsg()` or `WinPeekMsg()` and there are no other messages in the queue, if there are any windows associated with the current queue that require updating (non-empty update regions), a `WM_PAINT` message is returned for that window. The `WM_PAINT` message is not actually placed in the application queue; they are returned only as often as `WinGetMsg()` or `WinPeekMsg()` is called. Since multiple invalidations are accumulated into the single update region, a single `WM_PAINT` message may be generated as a result of more than one invalidation.

`WM_PAINT` messages are always generated in a top-down fashion: first the parent, then its children, etc.

If necessary, an application may ensure that an async paint window has been updated by calling the `UpdateWindow()` function. If the window has an update region, a `WM_PAINT` message is sent to the window procedure.



#### *4.1.2.8.2 WM\_PAINT Message Processing*

The WM\_PAINT message is processed by calling the WinBeginPaint() function. The WinBeginPaint() function returns a PS handle that is associated with a window DC with a visible region set to the intersection of the window's visible area and its update region: this is the area that must be updated. All drawing to the PS will only be done in those areas that require updating.

If the window is using a cached PS, then the returned PS handle is from the cache. If a window DC was created for the window, the PS handle to use for updating is passed as a parameter to BeginPaint.

With the PS handle, the window procedure must simply redraw the contents of the window. Since the PS is clipped to the update region, only the part of the window that needs to be updated is redrawn.

The rcPaint rectangle is the bounding rectangle (in window coordinates) of the area that must be updated. This rectangle can be used to minimize the amount of redrawing that must be done.

After the window has been repainted, WinEndPaint() is called to restore the visible region of the window DC to its previous state. If the PS handle is a cached PS, then the PS is released by WinEndPaint().

#### *4.1.2.8.3 Incremental Window Updating*

It is sometimes useful to update an async paint window incrementally one part at a time. The idea is that multiple WM\_PAINT messages are required to repaint the entire window. This is especially useful with windows that paint slowly: after processing each WM\_PAINT message, execution returns back to the application's main loop, where user input can be processed if it has occurred.

This can be accomplished by calling either WinGetUpdateRect() or WinGetUpdateRgn() to obtain a rectangle or region that describes the area that needs updating. These calls do not remove the window's update region. Based on this information, a part of the window can be updated, and validated with WinValidateRgn() or WinValidateRect().

#### **4.1.2.9 Synchronous Window Updating**

If a window rearrangement or window showing or hiding operation occurs, any affected windows with the CS\_SYNCPAINT style are updated before the operation is completed. Windows with the CS\_SYNCPAINT style are called "Sync Paint" windows.

Sync paint windows are usually painted as soon as they are invalidated. If the sync paint window has an async parent that isn't `WS_CLIPCHILDREN` and they both get invalidated by a single invalidation, the painting of its sync paint children is deferred until the async parent processes the `WM_PAINT` message, so that the correct drawing order (top-down, parent-child) is maintained.

These messages are also sent as a result of calls to `WinInvalidateRect()` or `WinInvalidateRgn()`, and `WinEndPaint()`.

#### 4.1.2.10 Synchronous vs. Asynchronous Painting

Synchronous painting windows are well suited when a window can (or must) be drawn very quickly, as is the case with the controls that make up a window frame or a dialog box. Sync paint windows must draw quickly because the operation that is causing the `WM_PAINT` message will not complete until the message has been processed.

Asynchronous painting is appropriate for windows that are relatively slow to be redrawn. By putting off redrawing until there is less activity, the system will keep up with a user that types or uses the mouse quickly. This feature can prevent the swapping in and out of application code as well.

Generally, asynchronous painting is used for main application windows.

### 4.1.3 Window Drawing Functions

This section documents the calls mentioned above plus several drawing helpers. It is assumed that you've read the overview above.

---

#### WinGetPS

---

##### Format

```
HPS WinGetPS(hab, hwnd)
HWND hwnd;
HAB hab;
```

##### Description

This function returns a PS handle that can be used for drawing in the specified window. The returned PS is a "micro-PS"; not all GDI calls may be made. The initial state of the PS is the same as the initial state of a PS created with `GdiCreatePS()`.

The visible region of the returned PS depends on the existence of the following window and class styles:

---

Style	Description
WS_CLIPCHILDREN	All of the window's children are excluded.
WS_CLIPSIBLINGS	All siblings above the window are excluded.
CS_PARENTCLIP	<p>The visrgn is the same as the window's parent. The PS origin and pattern drawing origin is established normally.</p> <p>This style optimizes the use of the PS cache by minimizing the visrgn calculation required for child windows.</p>

---

---

### WinReleasePS

---

#### Format

```
BOOL WinReleasePS (hps)
HPS hps;
```

#### Description

This function returns a PS handle obtained with WinGetPS() to the cache. Returns TRUE if successful, FALSE otherwise.

See section on "Cached PS's" at the beginning of this section.

---

---

### WinOpenWindowDC

---

#### Format

```
HDC WinOpenWindowDC (hwnd)
HWND hwnd;
```

#### Description

Opens a window DC associated with the specified window handle that may be used with GpiCreatePS() or GpiAssocDC() in order to obtain a PS to draw in the window. Returns NULL if unsuccessful.

#### Notes

The window DC is automatically destroyed when its associated window is destroyed. The returned DC handle must NOT be destroyed with

GpiDestroyDC().

The visrgn of the DC is updated automatically as windows are rearranged.

Only one window DC per window may be created.

---

## WinBeginPaint

---

### Format

```
HPS WinBeginPaint(hwnd, hps, lprcPaint)
HWND hwnd;
HANDLE hps;
LPRECT lprcPaint;
```

### Description

This function is called during the processing of the WM\_PAINT message in order to obtain a PS handle with a visible region corresponding to the area of the window that should be repainted.

If hps is NULL, one is obtained for use from the PS cache. Otherwise, the window DC associated with hwnd is selected into hps, and hps is returned.

If lprcPaint is not equal to NULL, this function also fills in the rectangle pointed to by lprcPaint with the rectangle bounding the window area needing updating, in window coordinates (regardless of the transform of the hps).

### Notes

The update region associated with hwnd is removed; i.e., the entire window is validated.

WinBeginPaint() hides the caret if it is flashing in hwnd, and later shows it again in WinEndPaint(), which must be called once the application is finished drawing.

See "Asynchronous Window Updating" at the beginning of this section.

---

## WinEndPaint

---

### Format

```
void WinEndPaint(hps)
HPS hps;
```

### Description

This function is used at the end of WM\_PAINT message processing to restore the PS used for

repainting to its original state, and to update the invalid sync paint windows of hwnd.

If the PS used for repainting was associated with a different DC handle than the window DC selected by WinBeginPaint(), the previous DC handle is reassociated with the PS.

If the caret was hidden by WinBeginPaint(), the caret is shown again.

Note that if the window for which the WinBeginPaint() / WinEndPaint() sequence is done has some Synch Paint children, these are automatically updated by the WinEndPaint call.

See "WinBeginPaint()" above and "Asynchronous Window Updating" at the beginning of this section.

---

## WinInvalidateRect

---

### Format

```
BOOL WinInvalidateRect(hab, hwnd, lprc)
HWND hwnd;
LPRECT lprc;
HAB hab;
```

---

## WinInvalidateRgn

---

### Format

```
BOOL WinInvalidateRgn(hab, hwnd, hrgn)
HWND hwnd;
HRGN hrgn;
HAB hab;
```

### Description

If the specified window is an asynchronously painted window, These function adds a rectangle or a region to the specified window's update region. If either lprc or hrgn is NULL, the entire window is invalidated.

If the window is a CS\_SYNCPOINT window, the window is repainted before WinInvalidateRect() returns.

If the window is WS\_CLIPCHILDREN and invalid area overlaps some CS\_SYNCPOINT children, those children will be repainted before WinInvalidateRect() returns.

If `hwnd` is `NULL`, area of of the screen (the desktop window) is invalidated. In this case, if `lprc` or `hrgn` is `NULL`, the entire screen is invalidated.

These functions return `TRUE` if successful, `FALSE` otherwise.

See "Asynchronous Window Updating" for more information.

---

### WinValidateRect

---

#### Format

```
BOOL WinValidateRect(hab, hwnd, lprc)
HWND hwnd;
LPRECT lprc;
HAB hab;
```

---

### WinValidateRgn

---

#### Format

```
BOOL WinValidateRgn(hab, hwnd, hrgn)
HWND hwnd;
HRGN hrgn;
HAB hab;
```

#### Description

These functions subtracts a rectangle or region from the specified window's update region. This function is only used with async update windows.

These functions have no effect on a window if any part of the window has been invalidated (such as as a result of another thread's window rearrangement) since the last time `WinBeginPaint()`, `WinGetUpdateRect()`, or `WinGetUpdateRgn()` was called.

These functions return `TRUE` if successful, `FALSE` otherwise.

See "Asynchronous Window Updating" for more information.

---

### WinQueryUpdateRect

---

#### Format

```
BOOL WinQueryUpdateRect(hwnd, lprc)
```

```
HWND hwnd;  
LRECT lprc;
```

#### Description

This function returns the rectangle that bounds the update region of `hwnd`. This routine is usually used as an alternate to `WinBeginPaint()` and `WinEndPaint()`, in incremental updating schemes. Typically the application calls `WinQueryUpdateRect()` to see what part of `hwnd` needs updating, updates it, then calls `WinValidateRect()` or `WinValidateRgn()` to subtract that updated part from the update region. `FALSE` is returned if `hwnd` is totally valid (no update region). `lprc` points to a buffer that receives the update rectangle in window coordinates.

See "Asynchronous Incremental Updating" for more information.

---

### WinQueryUpdateRgn

---

#### Format

```
INT WinQueryUpdateRgn (hwnd, hrgn)  
HWND hwnd;  
HRGN hrgn;
```

#### Description

This function copies `hwnd`'s update region into the already existing region passed in, `hrgn`. This routine is usually used as an alternate to `WinBeginPaint()` and `WinEndPaint()`, in incremental updating schemes. Typically the application calls `WinQueryUpdateRgn()` to see what part of `hwnd` needs updating, updates it, then calls `WinValidateRect()` or `WinValidateRgn()` to subtract that updated part from the update region. A code indicating the type of the region is returned, as for `GpiCombineRegion`.

The application can select `hrgn` into a `ps` as the clip region and have drawing clipped to the window's update region.

See "Asynchronous Incremental Updating" for more information.

---

### WinUpda

**Format**

```
BOOL WinUpdateWindow(hwnd)
HWND hwnd;
```

**Description**

This function forces the updating of a window and its children. If `hwnd` is an async window, it and only its async children are updated. They get sent `WM_PAINT` messages from within `WinUpdateWindow()`. If `hwnd` is a sync window, it and only its sync children are updated. They get sent `WM_PAINT` messages from within `WinUpdateWindow()`.

If `hwnd` is a child of a non-clip children parent, `hwnd`'s update region is subtracted from the update region of the parent, if the parent has one. This is so the parent, who will be drawing after `hwnd`, will not draw on top of what `hwnd` draws.

If the window was updated, this function returns `TRUE`, otherwise `FALSE`.

See "Visrgr Calculation" at the beginning of this section.

---

**WinExcludeUpdateRgn**

---

**Format**

```
INT WinExcludeUpdateRgn(hps, hwnd)
HPS hps;
HWND hwnd;
```

**Description**

This function subtracts the update region for `hwnd` (if one exists) from the visrgr of `hps`. This is used to prevent the application from drawing in areas of `hwnd` which are already invalid.

A code indicating the type of clipping area is returned, as for `GpiCombineRegion`.

**Notes**

This function is typically used in situations to avoid drawing in a window when it is likely most of the window is invalid, such as when replacing a selection when a window receives a `WM_SETFOCUS` message.



---

**WM\_PAINT**

---

**Format**

WM\_PAINT  
lParam1: 0  
lParam2: 0L

**Description**

This message is sent to asynchronously updated windows whenever Presentation Manager or an application makes a request to repaint a window's invalid bits. This message is either sent from WinUpdateWindow() when the application makes this call, or is returned from WinGetMsg(). In either case, the application only receives the message if the window has an update region.

See "Asynchronous Window Updating" at the beginning of this section.

---

**WinLockScreen**

---

**Format**

BOOL WinLockScreen(hab, fLock)  
HAB hab;  
BOOL fLock;

**Description**

WinLockScreen() is design to start and stop all screen output. If fLock is TRUE, nothing outputs to the screen until WinLockScreen() is called again with fLock FALSE. All the areas would have been drawn by applications while the screen is locked are remembered and updated once the screen is unlocked.

This function is used by threads that want to draw on an area of the screen in which they have no control. The user interface sizing and moving code uses WinLockScreen() while sizing or moving a window. All threads still run while the screen is locked.

This function does not prevent screen group switches (which may be required to handle a hard error.) If fLock is TRUE, this function always returns TRUE. If fLock is FALSE, this function returns TRUE if no screen group switch occurred, or if the screen is locked. Otherwise, returns

TRUE.

Note: If one thread locks the screen, other threads that call `LockScreen()` are blocked (although they can receive messages) until the first thread unlocks the screen.

---

## WinLockVisRgns

---

### Format

```
INT WinLockVisRgns(hab, fLock)
HAB hab;
BOOL fLock;
```

### Description

This function is called by a thread if it doesn't want any visrgns changing while it performs an operation on the screen, like copying screen bits into a memory bitmap. This function will block any other thread that tries to alter visrgns. While visrgns are locked, no messages should be sent, and no functions called that could send messages.

If `fLock` is `TRUE`, all visrgns are locked. If `fLock` is `FALSE`, all visrgns are unlocked.

More than one thread can concurrently lock visrgns. The system increments a lock count, each time an app makes a lock call, and decrements a count each time a thread makes an unlock call. The visrgns can not change unless the lock count is zero.

This function returns the lock count that results from the call.

**Note** An thread is not allowed to unlock visrgns if the corresponding lock was performed by another process.

### 4.1.3.1 Drawing Helpers

---

## WinScrollWindow

---

### Format

```
void FAR WinScrollWindow(hwnd, dx, dy,
                        lprcScroll, lprcClip, hrgnUpdate,
                        lprcUpdate, rgfsw)
HWND hwnd;
```

```
int dx;  
int dy;  
LPRECT lprcScroll;  
LPRECT lprcClip;  
HRGN hrgnUpdate;  
LPRECT lprcUpdate;  
UINT rgfsw;
```

#### Description

This routine scrolls the rectangle defined by \*lprcScroll in the window hwnd by dx units horizontally and dy units vertically. All coordinates must be in device units.

If lprcScroll is NULL, the entire window will be scrolled. \*lprcClip is a clip rectangle that clips the destination of the scroll. Any part of hwnd's update region that maps to scrolled bits will be offset too.

If not NULL, lprcUpdate is filled with the bounding region of the invalid bits uncovered by the scroll. If not NULL, hrgnUpdate is modified to hold the region uncovered by the scroll, in window coordinates.

rgfsw is an array of bits, which may be OR'ed together:

---

#### SW\_SCROLLCHILDREN

All children falling within the intersection of lprcScroll and lprcClip will be scrolled by dx and dy units.

#### SW\_INVALIDATERGN

The invalid region created as a result of the scroll will be added to update regions of those windows affected. This may result in the sending of WM\_PAINT messages to CS\_SYNCPAINT windows before WinScrollWindow() returns.

WinScrollWindow() returns a code indicating the type of invalid region created by the scroll, as returned by GpiCombineRegion. *Note:* If hwnd is not a clip children window, the bits of any child falling inside the scrolled area will be scrolled too. If this is the case, WinScrollWindow() should be called with SW\_SCROLLCHILDREN.

No thread should be moving bits around in its own window by any other method than by using WinScrollWindow(), due to the critical section nature of window update regions.

The fastest scrolling method is without `SW_SCROLLCHILDREN` and `SW_INVALIDATERGN`. If scrolling needs to repeat quickly, don't include the `SW_INVALIDATERGN` flag, and just repaint the invalid area if the invalid region is rectangular, otherwise invalidate and update.

If the scrolling does not happen often, include the `SW_INVALIDATERGN` flag, and `WinScrollWindow()` will invalidate and update sync paint windows automatically before returning.

---

## WinDrawText

---

### Format

```
INT WinDrawText(hps, lpchText, cchText, lprc, rgfCmd)
HPS hps;
LPSTR lpchText;
INT cchText;
LPRECT *lprc;
UINT rgfCmd;
```

### Description

This function draws a single line of formatted text in the rectangle specified by `lprc`. `hps` is a handle to a presentation space, `lpchText` is a far pointer to the character string to be drawn, `cchText` is the count of characters to be drawn, and `rgfCmd` is an array of flags specifying how to draw the text.

This function returns the actual number of characters drawn that fit completely within `lprc`.

If `cchText` is 0, the string is assumed to be zero terminated, and its length is automatically calculated by `WinDrawText()`.

The text is drawn using the currently selected text color and background color in the presentation space given by `hps`. The output is clipped to the rectangle specified by `lprc` unless the `DT_NOCLIP` command is used.

If a carriage return or line feed character occurs in the string it is assumed to terminate the line, even if the line is shorter than `cchText`.

`rgfCmd` may be any combination of the following values:

---

WinDrawText() Flags  
Flag

DT\_LEFT  
Left justify the text.

DT\_CENTER  
Center the text.

DT\_RIGHT  
Right justify the text.

DT\_VCENTER  
Vertically center the text.

DT\_TOP  
Top justify the text.

DT\_BOTTOM  
Bottom justify the text.

DT\_HALFTONE  
Halftone the text display.

DT\_MNEMONIC  
If a mnemonic prefix character is encountered, the next character is drawn with mnemonic emphasis.

DT\_GETTEXTENT  
No drawing is performed; \*lprc is changed to a rectangle that bounds the string if it were drawn with WinDrawText().

DT\_UBREAK  
Only words that fit completely within the supplied rectangle are drawn. Words are assumed to be separated by space characters.

Text is always drawn in the current font.

---

## WinHalftoneBitmap

---

Format

```

BOOL WinHalftoneBitmap(hps, hbm, hbr, xdst,
                        ydst, cxdst, cydst, xsrc, ysrc)
HPS hps;
HBITMAP hbm;
HBRUSH hbr;
INT xdst, ydst, cxdst, cydst;
INT xsrc, ysrc;

```

Description

Thus function outputs a halftoned bitmap at location `xdst`, `ydst`, with extents of `cxdst` and `cxdst` in the presentation space `hps`. `xsrc` and `ysrc` specify the starting location within the bitmap. The background of that location must be drawn first, as `WinHalftoneBitmap()` only draws the 'halftoned' bits.

The return value is `TRUE` if the operation was a success.

---

`WinDrawIcon`

---

Format

```
BOOL WinDrawIcon(hps, x, y, hIcon, rgfHalftone)
HPS    hps;
INT     x, y;
HICON   hIcon;
UINT    rgfHalftone;
```

Description

Where `hps` is handle to a presentation space. `x` and `y` are the coordinates at which to draw the icon. `hIcon` is a handle to the icon to draw. It may be a cursor handle. `rgfHalftone` is a word of flags consisting of the following icon styles which may be OR'd together:

---

`ICS_NORMAL`

draw the icon as it would normally appear

`ICS_HALFTONE`

draw the icon with a halftone pattern where black normally appears

`ICS_INVERT`

draw the icon inverted - ie black where white is and white where black is.

---

`WinDrawBorder`

---

Format

```
void far WinDrawBorder(hps, lprc, cx, cy, rgfCmd)
HPS hps;
LPRECT lprc;
INT cx;
INT cy;
```

UINT rgfCmd;

#### Description

This function draws a border (a rectangular frame) bounded by the rectangle \* lprc in the specified PS. cx is the width of the left and right sides of the rectangle, cy is the height of the top and bottom sides of the rectangle.

The values for rgfCmd are:

---

WinDrawBorder() Flags

DB\_ PATCOPY

The PATCOPY raster op is used

DB\_ PATINVERT

The PATINVERT raster op is used

DB\_ STANDARD

cx and cy are multiplied by the system SV\_ CXBORDER and SV\_ CYBORDER constants.

DB\_ DLGBORDER

A standard dialog border is drawn. If DB\_ PATCOPY specified, then an active dialog border is drawn. If DB\_ PATINVERT is specified, then an inactive dialog border is drawn.

DB\_ INTERIOR

Specifies that the interior of the border is drawn with the current pattern background color.

The border is drawn in the current pattern foreground color

#### Example

For example, here is a call to draw a rectangular frame whose width is twice the SV\_ CXBORDER and SV\_ CYBORDER values with the standard window frame color:

```
WinDrawBorder (hps, (LPRECT)&rc,  
                2, 2, DB_PATCOPY | DB_STANDARD);
```

---

#### WinInvertRect

---

##### Format

```
void WinInvertRect (hps, lprc)
```

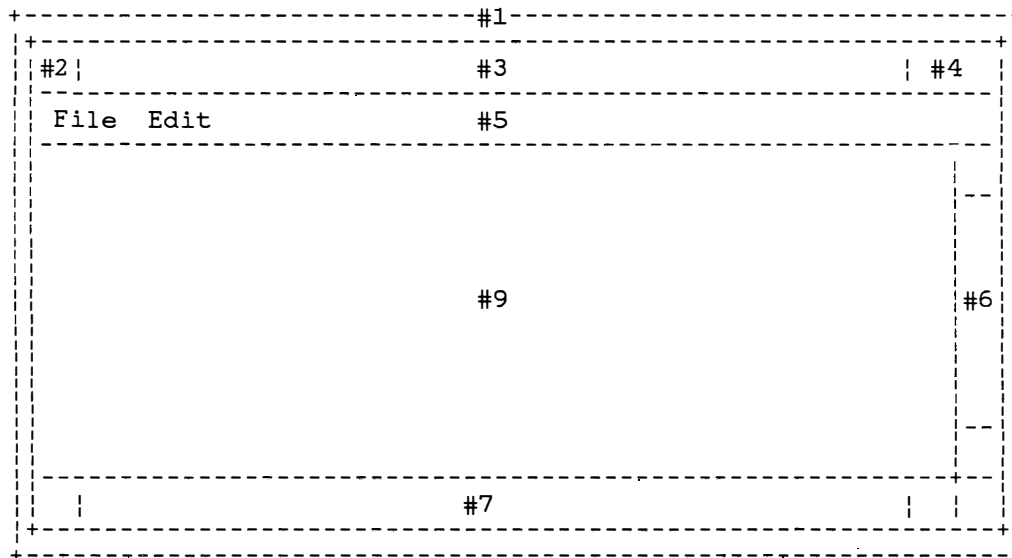
## Description

This function inverts the area of hps described by the rectangle \*lprc.

## 4.1.4 Window Frames

### 4.1.4.1 Window Frame Architecture

A standard application window has the following parts:



Part	Description
#1	Wide sizing border
#2	System menu
#3	Title Bar
#4	Minimize/Maximize box
#5	Application menu
#6	Vertical scroll bar
#7	Horizontal scroll bar
#8	Client window

The window that contains all of these parts is called the "Frame Window". Each of the parts that make up a window, such as the title bar and menu, are separate child windows of the frame window. All of these child windows (except the client area) are called "Frame Controls".



The client area is not a frame control; it is an instance of a window class implemented by the application.

The frame window and all of the frame controls are implemented with standard preregistered window classes. In this section, we will describe the behavior of each of these window classes, and how they are used with an application window class to implement a standard application window.

The frame window is the "glue" that holds together all of the frame controls and the client that make up an application window. It is responsible for arranging the frame controls and the client window as the frame window is sized and moved. It is also responsible for routing certain messages to its frame controls and the client window.

Each of the frame controls is known to the frame window by its window ID. Below is a list of the standard ID values that identify each frame control:

---

Standard Frame Control IDs

FID\_ SIZEBORDER

Wide sizing border

FID\_ SYSMENU

System menu

FID\_ TITLEBAR

Title Bar

FID\_ MINMAX

Minimize/Maximize box

FID\_ MENU

Application menu

FID\_ VERTSCROLL

Vertical scroll bar

FID\_ HORZSCROLL

Horizontal scroll bar

FID\_ CLIENT

Client window

As with all controls, when something interesting happens to a frame control (a scroll bar click or the menu key is pressed, for example), the frame control notifies its owner with window messages. The owner of each of the frame controls is the frame window itself. Messages that may be of interest to the client window are sent to the client by the frame window. The frame window has no owner.

There are two ways that you can create a standard window and frame controls. You can create the frame window and all of its frame controls explicitly, or you can use the `WinCreateStdWindow()` function, which will create a frame window plus all of the standard controls, using the frame window style to determine which of the standard frame controls are to be created.

#### **4.1.4.2 The Frame Window Class**

The standard window frame class name is `WC_FRAME`. The frame window is a sync paint window.

##### *4.1.4.2.1 Frame Window Styles*

When a window of class `WC_FRAME` is created, some of the standard frame controls may be also automatically created depending on the window style of the frame window. The window style also governs the appearance of the frame window border as well.

The frame window styles below are also used with `WinCreateStdWindow()` and `WinCreateFrameControls()` below.

---

Standard Frame Styles	
Flag	
<code>FS_TITLEBAR</code>	Title bar
<code>FS_SYSMENU</code>	System menu
<code>FS_MENU</code>	Application menu
<code>FS_MINMAX</code>	Minimize/Maximize box
<code>FS_VERTSCROLL</code>	Vertical scroll bar
<code>FS_HORZSCROLL</code>	Horizontal scroll bar
<code>FS_SIZEBORDER</code>	Wide sizing border
<code>FS_BORDER</code>	Window is drawn with a thin border

**FS\_DLGBORDER**

Window is drawn with a standard dialog border

**FS\_ACCELTABLE**

Causes an accelerator table to be loaded for this frame window from the resource file identified on the WinCreateStdWindow call.

**FS\_STANDARD**

Same as (FS\_TITLEBAR | FS\_SYSMENU | FS\_MINMAX | FS\_SIZEBORDER | FS\_MENU | FS\_ACCELTABLE)

All of the styles except FS\_BORDER and FS\_DLGBORDER only have an effect when the WC\_FRAME window is created. At this time, these styles are used to determine whether the corresponding frame control should be created. Once a frame control is created, changing these style bits has no effect: to add or remove frame controls the WinSetParent() call may be used, or the frame controls may be created or destroyed explicitly.

*4.1.4.2.2 Frame Window Messages*

The following messages are processed by the frame window. Many of these messages are standard window management messages; they are included here to illustrate the behavior of the frame window.

---

**WM\_SYSCOMMAND**

Description of this message is given in the section on Controls.

---



---

**WM\_FORMATFRAME**

---

**Format**

```
WM_FORMATFRAME
lParam1:      OL;
lParam2:      OL
Returns:      BOOL fProcessed;
```

**Description**

This message is sent to a frame window to calculate the sizes and positions of all of the frame controls and the client window.

This message is passed on to the FID\_CLIENT window by the standard frame window. If the FID\_CLIENT window returns fProcessed == FALSE, then the frame window performs the default action; otherwise it is assumed that the FID\_CLIENT window processed the message.

The `WC_FRAME` default processing of this message is to first call `WinFormatFrame()` and then calling `WinSetMultWindowPos()` to position the frame controls.

---

## WM\_UPDATEFRAME

---

### Format

```
WM_UPDATEFRAME
lParam1:      ULONG style;
lParam2:      OL
Returns:      BOOL fProcessed;
```

### Description

This message is sent by an application after frame controls have been added or removed from the window frame to reformat and update the appearance of the window frame as a result of the change.

`lParam1` contains `FS_` style bits that indicate which frame controls were added or removed.

Since this message will cause any redrawing that is necessary, you should ensure that no drawing takes place when you actually add or remove a frame control, to prevent unnecessary redrawing. If you use `WinSetParent()`, this is done by setting the `fRedraw` parameter to `FALSE`.

This message is passed on to the `FID_CLIENT` window by the standard frame window. If the `FID_CLIENT` window returns `fProcessed == FALSE`, then the frame window performs the default action; otherwise it is assumed that the `FID_CLIENT` window processed the message.

The default `WC_FRAME` processing of this message is to simply send a `WM_FORMATFRAME` message to itself.

To add a control, simply create the window with a zero size, or use `WinSetParent()` with `fRedraw == FALSE`. To remove a control, use `WinSetParent()` with `fRedraw == FALSE`. Then set the appropriate style bit, and send the `WM_UPDATEFRAME` message to the frame window.

---

## WM\_ERASEBACKGROUND

---

## Format

```
WM_ERASEBACKGROUND
lParam1:      HPS hpsFrame;
lParam2:      LPRECT lprcPaint;
Returns:      BOOL fProcessed;
```

## Description

This message is sent by WC\_FRAME and WC\_DIALOG windows to the FID\_CLIENT window in order to allow the client window to erase the background of the frame window synchronously.

- hpsFrame is the HPS for the frame window.
- lprcPaint contains a far pointer to a rectangle to be painted.

If the client window processes the message, TRUE is returned. If FALSE is returned and a FID\_CLIENT window exists, then the area of the frame covered by the FID\_CLIENT window is erased in the system window background color. If FALSE is returned and no FID\_CLIENT window exists, then the entire frame window is erased in the system window background color.

#### *4.1.4.2.3 Default WC\_FRAME Processing of Standard Messages*

Here is a list of the default processing of various messages by WC\_FRAME class windows:

---

**WM\_ACTIVATE**

First sends TBM\_SETSTATE message to FID\_TITLEBAR control, if it exists, to hilite or unhilite the titlebar. If FS\_DLGBORDER then dlg border is redrawn in either hilited or unhilited state, as necessary. Next sends the WM\_ACTIVATE message to the FID\_CLIENT window. Upon return, if the window is being deactivated and the caret is set to the frame window or any of its controls, the caret is destroyed.

**WM\_CALCVALIDRECTS**

Passed to WinDefWindowProc()

**WM\_SIZE:**

Sends a WM\_FORMATFRAME message to self.

**WM\_FORMATFRAME**

First, the message is sent to the FID\_CLIENT window. If FID\_CLIENT returns TRUE to indicate that the message was processed by the client window, then no additional

processing is performed. Otherwise, `WinFormatFrame()` is called, with `lprcFrame` rectangle equal to the frame window rect adjusted based on the presence of the `FS_DLGBORDER` or `FS_BORDER` controls, or the existence of the `FID_SIZEBORDER` control. When `WinFormatFrame()` returns, `WinSetMultWindowPos()` is called to reposition the frame controls.

#### WM\_UPDATEFRAME

First, the message is sent to the `FID_CLIENT` window. If `FID_CLIENT` returns `TRUE` to indicate that the message was processed by the client window, then no additional processing is performed. Otherwise, If `lParam1` contains `FS_SIZEBOX`, the area occupied by the size box is invalidated. A `WM_FORMATFRAME` is sent to self.

#### WM\_QUERYWINDOWPARAMS

Passed to `FID_TITLEBAR` control, if it exists, in order to obtain the text of the window title.

#### WM\_SETWINDOWPARAMS

Passed to `FID_TITLEBAR` control, if it exists, in order to set the text of the window title.

#### WM\_L/M/RBUTTONDOWN

`WinSetActiveWindow()` is called, with `fSetFocus == TRUE`.

#### WM\_COMMAND

Passed to `FID_CLIENT` window

#### WM\_HSCROLL

Passed to `FID_CLIENT` window

#### WM\_VSCROLL

Passed to `FID_CLIENT` window

#### WM\_PAINT

If `FS_BORDER` then thin frame is drawn. If `FS_DLGBORDER` then dialog border is drawn in either hilited or unhilited state, as necessary. `WM_ERASEBACKGROUND` message is sent to `FID_CLIENT` window. If `WM_ERASEBACKGROUND` message returned `FALSE` (client did not process), then client area is erased with the standard window background color.

#### WM\_SYSCOMMAND

Below is a list of the standard system commands and how they are handled by the frame processing:

---

System Command Values  
Command

#### SC\_SIZE

Send a `SZM_SIZE` message to the control with the ID `FID_WIDESIZE`, assumed to be the wide sizing

control.

#### SC\_MOVE

Send a CPM\_MOVE message to the control with the ID FID\_TITLEBAR, assumed to be the title bar control.

#### SC\_MINIMIZE

Minimizes the frame window or restores it to a remembered size and position.

#### SC\_MAXIMIZE

Maximizes the frame window or restores it to a remembered size and position.

#### SC\_NEXT

Cycle the active window status to the next top-level window.

#### SC\_APPMENU

Send a MM\_STARTMENU message to the control with the ID FID\_APPMENU, assumed to be the application menu control.

#### SC\_SYSMENU

Send a MM\_STARTMENU message to the control with the ID FID\_SYSMENU, assumed to be the system menu control.

#### 4.1.4.2.4 Frame Notifications

The frame window always sends notification messages to the child window that has the window ID of FID\_CLIENT. If no window exists with that ID, no notification messages are sent.

All messages that are not explicitly processed by the frame window are simply sent on to the FID\_CLIENT window. This includes the WM\_COMMAND message and the WM\_HSCROLL and WM\_VSCROLL messages posted by the menu and scroll bar frame controls, respectively.

---

#### WM\_QUERYMINMAXINFO

---

##### Format

```
WM_QUERYMINMAXINFO
lParam1:    POINT lprgptMinMax{3};
lParam2:    0;
Returns:    OL;
```

##### Description

This message is sent by the frame window to the client window before it performs a minimize or a

maximize operation, upon receiving a WM\_SYSCOMMAND message with lParam1 equal to either SC\_MINIMIZE or SC\_MAXIMIZE.

lParam1 is a far pointer to an array of 3 POINT structures. These points should be set to the values indicated below:

```
rgptMinMax{0} = Minimized size
rgptMinMax{1} = Maximized size
rgptMinMax{2} = Maximized position, relative
                 to the parent of the MinMax control
                 owner window.
```

#### 4.1.4.3 Standard Window Frame Routines

---

##### WinCreateStdWindow

---

###### Format

```
HWND FAR PASCAL WinCreateStdWindow(hab, hwndParent,
                                   style, lpszTitle, lpszClientClass, styleClient,
                                   idModule, idResource, lphwndClient)
HWND hwndParent;
ULONG style;
LPSTR lpszClientClass;
LPSTR lpszTitle;
ULONG styleClient;
UINT idModule;
INT idResource;
HWND far *lphwndClient;
HAB hab;
```

###### Description

WinCreateStdWindow() creates and returns a WC\_FRAME class window whose parent is hwndParent. The frame window handle is returned, or NULL if the creation was unsuccessful.

style is the window style of the WC\_FRAME window, which is a combination of any of the standard WS\_ window styles and the FS\_ frame styles.

lpszClientClass is a pointer to the class name of a window class name; if non-NULL, a client window is created of the specified class and with the window style styleClient, and returned in \*lphwndClient. Generally, WS\_VISIBLE should be included in styleClient.

If FS\_TITLEBAR is specified, a WC\_TITLEBAR control is created with the text pointed to by



lpszTitle. If FS\_ TITLEBAR is not specified, this parameter is ignored.

If FS\_ MENU is specified, idModule and idResource are the module handle, returned by the DOS DosLoadModule call, and ID of the menu template and accelerator table resources to load. If FS\_ MENU is not specified, idModule and idResource are ignored. Note that it is the application's responsibility to ensure that the ID of the menu resource and accelerator table resource are the same.

All of the frame controls and the client window are created with the standard FID\_ window IDs.

If WS\_ VISIBLE is specified in style, the standard window's size and position are obtained from the Shell before the window is shown. (The WinQueryTaskSizePos function is used).

If WS\_ VISIBLE is NOT specified in style, the frame window is created invisible with a zero size and positioned at bottom left. You MUST later size, position, and show the window with a call to WinSetWindowPos(). This is the recommended way to create a standard window which is NOT a main window.

## WinCreateStdWindowIndirect

---

### Format

```
HWND hwndFrame = WinCreateStdWindowIndirect(hab,  
                                              hwndParent, style, lpszClientClass, lpszTitle,  
                                              styleClient, lpMenuTemplate, lpAccelTable,  
                                              lpHwndClient)  
HWND hwndParent;  
ULONG style;  
LPSTR lpszClientClass;  
LPSTR lpszTitle;  
ULONG styleClient;  
UCHAR FAR *lpMenuTemplate;  
UCHAR FAR *lpAccelTable  
HWND far *lphwndClient;  
HAB hab;
```

### Description

This function is identical to WinCreateStdWindow, except that the lpMenuTemplate parameter is a pointer to a menu template and the lpAccelTable parameter is a pointer to an in-memory accelerator table definition. These parameters replace the idModule, returned by the DOS

DosLoadModule call, and idResource parameters of WinCreateStdWindow(). A NULL value implies that no menu or accelerator table is present, respectively.

### WinCalcFrameRect

---

#### Format

```
void WinCalcFrameRect (hwndFrame, lprc, fClient)
HWND hwndFrame;
LPRECT lprc;
BOOL fClient;
```

#### Description

This function is used to calculate a client rectangle from a frame rectangle, or a frame rectangle from a client rectangle, depending on the fClient flag. The rectangle at \*lprc is modified.

If fClient is FALSE, \*lprc points to a client rectangle. \*lprc is modified such that it corresponds to a size and position of hwndFrame that would result in a client window rectangle the same as that originally pointed to by lprc.

If fClient is TRUE, \*lprc points to a frame rectangle. \*lprc is modified such that it corresponds to the client window rectangle if the size and position of the hwndFrame were changed to the rectangle originally pointed to by lprc.

This function works even if hwnd is hidden. In fact, it is important that hwnd should be hidden if it is required that the window show a particular client rectangle when the window is first shown.

### WinCreateFrameControls

---

#### Format

```
BOOL FAR PASCAL WinCreateFrameControls (hwndFrame,
                                         style, lpszTitle, idModule, idResource)
register HWND hwndFrame;
ULONG style;
LPSZ lpszTitle;
UINT idModule;
INT idResource;
```

#### Description

This function creates all of the standard frame controls for a window, based on the style parameter. All of the controls are created with hwndFrame as their owner and parent. If FS\_ TITLEBAR is specified in style, a

WC\_TITLEBAR window is created with the text pointed to by lpszTitle. If FS\_MENU is specified, a menu is loaded from the resource template specified by idModule, returned by the DOS DosLoadModule call, and idMenu.

All of the controls are created with the standard FID\_ window IDs.

This function is typically used when the standard frame controls are to be used with a non-standard (i.e., non-WC\_FRAME) frame window class.

## WinFormatFrame

---

### Format

```
INT FAR PASCAL WinFormatFrame (hwndFrame,
                                lprcFrame, lprgswp, cswpMax, lprcClient)
HWND hwndFrame;
LPRECT lprcFrame;
LPSWP lprgswp;
INT cswpMax;
LPRECT lprcClient;
```

### Description

This function calculates the size and position of all of the standard frame controls within a frame window. For all of the standard frame control children of hwndFrame that exist, (identified by their FID\_ window ID values), this function fills in an array of SWP structures with the window size and positions. This array of SWP structures is then typically passed to WinSetMultWindowPos() to actually move and size the frame controls to the desired locations.

hwndFrame is the frame window handle whose children are to be positioned.

lprcFrame is a far pointer to a rectangle within which to format the children. This is typically the window rectangle of hwndFrame, but in cases where the frame window has a wide border (FS\_DLGBORDER for example), this rectangle is inset by the size of the border.

lprgswp is a far pointer to an array of SWP structures, and cswpMax is the maximum number of SWP structures that will fit in this array. This array should have at least 12 elements.

This function returns the actual number of SWP structures returned in the array. The array is filled in in the order of FID\_ values shown in the

table; the SWP structure for the FID\_CLIENT window is always the last element of the array. The client rectangle is returned in \*lprcClient. This rectangle is in frame window coordinates, and represents the area occupied by the FID\_CLIENT window. If lprcClient is NULL, no rectangle is returned.

This function is typically used only by applications that wish to have a non-standard window frame layout.

#### **4.1.4.4 Using Frame Windows**

To add or remove a menu, scroll bar, or other frame control, see the WM\_UPDATEFRAME message.

You can use WinLoadMenu() to load more than one menu, and use WinSetParent() and WM\_UPDATEFRAME to switch between the loaded menus.

When sizing, moving, or activating an application window, the frame window handle should be used. The client window handle should be used for things that directly affect the client window: drawing, sending messages, etc.

#### **4.1.4.5 Alternate Window Frame Formatting**

By processing the WM\_FORMATFRAME and WM\_UPDATEFRAME messages, it is possible for applications to change the default window frame formatting.

WM\_FORMATFRAME processing is usually done by first calling WinFormatFrame(), and changing or adding additional SWP structures to the SWP array based on the results of WinFormatFrame(). Then, WinSetMultWindowPos() is called with the changed SWP array.

### **4.1.5 The Title Bar Control**

The title bar control is the frame control that is used to display the application window title. It is also used to display the window active/inactive status of the frame window, and to implement window flashing.

The title bar control also implements the user interface for moving the frame window.

The class name to use when creating title bar controls is `WC_TITLEBAR`. The standard ID for a title bar control in a frame window is `FID_TITLEBAR`.

#### 4.1.5.1 Title Bar Style

There is only one title bar style; thus there are no available style bits.

#### 4.1.5.2 Title Bar Messages

The window text of the title bar control is displayed in the title bar; the window text may be accessed via the `WM_SETWINDOWPARAMS` and `WM_QUERYWINDOWPARAMS` messages.

The following messages are processed by the title bar control, in addition to the standard window management messages:

Title bar state flag constants:

---

State Flag.	Meaning
<code>TBF_FLASH</code>	Titlebar state is flashing.
<code>TBF_FLASHHILITE</code>	Titlebar is flashing and hilit.
<code>TBF_HILITE</code>	Titlebar state is hilit.

---

---

Format
<code>TBM_QUERYSTATE</code>
<code>lParam1:</code> <code>OL</code>
<code>lParam2:</code> <code>OL</code>
<code>Returns:</code> <code>UINT rgfTitleState;</code>

Description

Returns a combination of the `TBF_` constants above:

- `TBF_FLASH` if the titlebar is currently flashing

- TBF\_FLASHHILITE if titlebar is flashing and in hilited state
- TBF\_HILITE if titlebar is in hilited (active) state.

---

## TBM\_SETSTATE

---

### Format

```
TBM_SETSTATE
lParam1:    UINT  rgfNewState;
lParam2:    UINT  rgfStateMask;
Returns:    UINT  rgfOldState;
```

### Description

Changes the state of the title bar. lParam1 contains flags selecting the new state, and lParam2 contains flags indicating which state will be selected. For example, rgfNewState = TBF\_FLASH, rgfStateMask == TBF\_FLASH starts the window flashing. rgfNewState = 0, rgfStateMask == TBF\_FLASH | TBF\_HILITE unhilites the title bar and stops it flashing.

Returns the previous state (ANDed with rgfStateMask) of the title bar.

---

## TBM\_TRACKMOVE

---

### Format

```
TBM_TRACKMOVE
lParam1:    BOOL   fMouse;
lParam2:    POINT  ptMouse;
Returns:    OL;
```

### Description

Initiates the window movement user interface code. The owner window of the title bar is moved.

If fMouse is TRUE, the movement was initiated with the mouse, and lParam2 contains the initial mouse position. Otherwise, lParam2 contains 0L.

The processing of this message causes a WM\_QUERYMOVESIZEINFO message to be sent to the control owner. The processing of this message also results in a call to SetWindowPos.

### 4.1.5.3 Title Bar Notification Messages

---

#### WM\_QUERYMOVESIZEINFO

---

##### Format

```
WM_QUERYMOVESIZEINFO
lParam1:    LPRECT *lprcTrackBoundry;
lParam2:    POINT FAR lprgptSize[2];
Returns:    BOOL fContinue;
```

##### Description

This message is sent by the title bar control before moving, or by the size control before sizing. It is used to obtain limiting widths and rectangles that limit the sizing or moving operations.

For the title bar, this message is sent once to the title bar owner at the start of the processing of the `TM_TRACKMOVE` message.

For the size control, this message is sent once to the size control owner at the start of the processing of the `SZM_TRACKSIZE` message.

`lParam1` is a far pointer to a rectangle, which is initialized to the tracking limit rectangle. When the window is being moved, no edge of the window may fall outside this rectangle.

`lParam2` is a long pointer to an array of two `POINT` structures. `lprgptSize[0]` is set to the minimum tracking size, and `lprgptSize[1]` is set to the maximum window tracking size.

Returns `TRUE` to continue tracking, `FALSE` to abort tracking.

### 4.1.6 The Size Control

The size control is used to implement the wide window sizing borders. This controls implement the window sizing user interface. After a size change is requested, size controls notify their owners with an `SZM_SETPOS` message.

The class name to use when creating size controls is `WC_SIZE`. The standard IDs for the size borders when used with the standard `WC_FRAME` window class are `FID_WIDESIZE`.

#### 4.1.6.1 Size Control Styles

There is only one Size Control style so there are no available style bits.

#### 4.1.6.2 Size Control Messages

---

##### SZM\_TRACKSIZE

---

###### Format

```
SZM_TRACKSIZE
lParam1:    BOOL fMouse;
lParam2:    POINT ptMouse;
Returns:    OL;
```

###### Description

Initiate the window sizing user interface code. The owner window of the size control is moved.

If fMouse is TRUE, the size operation was initiated with the mouse, and lParam2 contains the initial mouse position. Otherwise, lParam2 contains 0L.

The processing of this message causes a WM\_GETSIZEINFO message to be sent to the control owner.

#### 4.1.6.3 Size Notification Codes

The only notification code is the WM\_QUERYMOVESIZEINFO message which is described in the Title Bar Notification messages section.

### 4.1.7 The Minimize/Maximize Control

The Minimize/Maximize controls (MinMax controls) are used as input translation controls, which simply translate mouse down messages into WM\_SYSCOMMAND messages that get sent to the owner window. When drawn, they typically appear as small icon buttons at the right edge of the title bar of the frame window.

The class name to use when creating min/max button controls is WC\_TITLEBAR. The standard ID for the minimize and maximize button control in a frame window is FID\_MINMAX.



#### 4.1.7.1 MinMax Control Styles

---

MinMax Control Styles  
Style

MMS\_MAXBUTTON

With this style the control provides the user interface for maximizing. Can be used with MMS\_MINBUTTON.

MMS\_MINBUTTON

With this style the control provides the user interface for maximizing. Can be used with MMS\_MINBUTTON.

If both the MMS\_MAXBUTTON and MMS\_MINBUTTON styles are used, the control appears and operates as both a min and a max button, with the min button to the left of the max button.

When clicked on, the min/max control simply sends a WM\_SYSCOMMAND message to the owner window. If the owner window is the frame window, it'll send a WM\_GETMINMAXINFO message to the client window, and based on the result, perform the minimizing or maximizing operation.

If a minimize button is clicked on, a WM\_SYSCOMMAND will be sent to the owner window, with lParam1 equal to SC\_MINIMIZE.

If a maximize button is clicked on, a WM\_SYSCOMMAND will be sent to the owner window, with lParam1 equal to SC\_MAXIMIZE.

#### 4.1.8 Dialog Boxes

A dialog box is a window that contains one or more child windows, typically used to provide a means for an application to gather input from the user. It is often a temporary window that is created for special purpose input and destroyed immediately after use.

There are two types of dialog boxes: modeless dialogs and modal dialogs. A modeless dialog allows other application windows to be activated after it has been created. A modal dialog keeps control until the WinDismissDlg function is called. The user is not able to activate other windows belonging to the same application until he has finished interacting with the modal dialog.

Dialog boxes are created from a "Dialog Template". The dialog template defines the position, appearance, and window ID of the dialog window, and each of its child windows. Dialog Templates can be used to create dialog windows of any window class, containing controls of any window class. For standard dialog boxes, the dialog window itself is created with the WC\_DIALOG class, and its children are any of the preregistered control

classes. Applications can create dialogs with application registered controls as well.

Standard `WC_DIALOG` dialog boxes also have a "Dialog Procedure". The dialog procedure is identical to any normal window procedure. Presentation Manager always calls the dialog procedure, giving it a chance to process the message. If the dialog procedure does not need to handle the message, it generally passes it on to `WinDefDlgProc()`. `WinCreateDlg()` automatically performs the "sub-classing" that is required. Note that when a dialog box is created with `WinCreateDlg()`, the dialog proc won't receive a `WM_CREATE` message. This is because the dialog window does not get sub-classed until AFTER it is created.

Presentation Manager supports a standard user interface for dialog boxes. Generally, when a dialog proc doesn't handle a keydown or mouse button message, it must return `FALSE`. Presentation Manager then performs additional processing on this message to implement the user interface.

The dialog window itself is called the "Dialog Window", and the child control windows are called "Dialog Items". Each dialog item is identified by a unique ID that is passed to `WinCreateWindow()` as the item is created. The window handle of a particular item may be obtained with `WinWindowFromID()`.

As each dialog item window is created, its text is processed with `WinSubstituteStrings()`, which may cause `WM_SUBSTITUTESTRING` messages to be sent to the dialog procedure. See "WinSubstituteStrings" for more information.

#### 4.1.8.1 The Dialog Procedure

A dialog procedure is a normal window procedure that is automatically sub-classed to each instance of the `WC_DIALOG` window class. The dialog procedure must be declared as follows:

---

DialogProc

---

Format

```
ULONG FAR PASCAL DialogProc(hwnd, msg,  
                             lParam1, lParam2)  
HWND  hwnd;  
UINT  msg;  
ULONG lParam1;  
ULONG lParam2;
```

This procedure is no different from other window procedures except that it can receive predefined window messages intended especially for dialog boxes. It won't receive a `WM_CREATE` message, but it can get the same information from the `WM_INITDIALOG` message.

The first four parameters are the same as with any window procedure. `hwnd` is always the window handle of the dialog box window.

Typically, the dialog procedure will process some, but not all of the messages passed to it. If it does not wish to process a message, it should pass it on to `WinDefDlgProc()`.

#### 4.1.8.2 Dialog Templates

Dialog Templates are data structures used to define dialog boxes. These templates can be loaded as resources or created dynamically in memory. Dialog Templates can be used to create windows of any window class that contains child windows of any window class. For standard dialog boxes, the dialog window itself is created with the `WC_DIALOG` class, and its children are any of the predefined control classes.

The dialog template specifies all the information required to create a dialog window and its children: the class, text, position, size, and the window ID.

##### 4.1.8.2.1 Dialog Coordinates

Coordinates in a dialog template are specified in "dialog coordinates". Dialog coordinates are based on the size of the system font: a unit in the horizontal direction is 1/4 the system font average character width, and a unit in the vertical direction is 1/8 the system font character height. The origin is the bottom left corner of the dialog box.

##### 4.1.8.2.2 Dialog Template Format

Dialog Template format:

```
typedef struct
    UINT          cbTemplate;
    UINT          type;
    UINT          codepage;
    UINT          offrgti;
    UINT          statusTemplate;
    UINT          coffPresParams;
    UINT          rgoffPresParams[ coffPresParams ] ;
    DLGTITEM      rgti[ ? ] ;
    UCHAR         rgbData[ ? ] ;
DLGTEMPLATE;
```

cbTemplate	is the overall length of the dialog template in bytes.
Type	identifies the dialog template format type. This value is <i>currently</i> always zero.
codepage	The codepage of the text in the template. This is currently always 850.
offrgti	Offset to array of DLGITEM structures from beginning of template.
statusTemplate	contains status information for the entire dialog box. This currently always has a value of zero.
coffPresParams	Currently always zero
rgoffPresParams	Currently always an array of zero dimension
rgti	Array of DLGITEM structures, described below. The first element of this array describes the primary window. All other items in the array are children of this window. The number of children is specified in the cChildren field of the first DLGITEM.
rgbData	Array of bytes. Window class names, text strings, control data and presentation parameters are stored here.

Dialog Item format:

```
typedef struct DLGITEM
{
    UINT    statusItem;
    UINT    cChildren;
    UINT    cchClassName;
    UINT    offClassName;
    UINT    cchText;
    UINT    offText;
    ULONG   Style;
    INT     x;
    INT     y;
    INT     cx;
    INT     cy;
    UINT    id;
    UINT    ioffPresParams;
    UINT    offCtlData;
} DLGITEM;
```

- statusItem**  
contains status information for the item. This must have the value 0.
- cChildren**  
is a count of the child windows that are owned by this window. If cChildren is non-zero, then the next cChildren windows will be created as children to this window. Each window can have any number of child windows, which allows for a fully tree-structured arrangement.
- cchClassName**  
is the length of window class name. If a length of zero is specified then ClassOffset is assumed to contain a system class identifier. This is the length of the significant characters in the class name **excluding the terminating NULL**.
- offClassName**  
is the offset within the dialog template of the string specifying the window class name. If a cchClassName of zero is specified then this offset is assumed to be a system class identifier. The class name should be a NULL terminated ASCII string.
- cchText** is the length in bytes of the text data associated with the control. This is the length of the significant characters in the text **excluding the terminating NULL**.
- offText** is the offset within the dialog template of the text data with which the window should be initialized. The text should be a NULL terminated ASCII string.
- Style** is the style of the window. The first 8 bits are the standard WS\_ style bits, and the remaining 24 bits are available for class-specific use.
- X and Y**  
are the x and y coordinates of the window. For the control windows, within the dialog these are specified in dialog coordinates, with x and y relative to the origin of the dialog window.
- Cx and Cy**  
are the x and y extent of the window.
- id** is the ID number for this window.
- ioffPresParams**  
Currently always zero.
- offCtlData**  
is the offset within the dialog template of the Control data for this window.

The Dialog Data contains the data for the Text, Class Names, CreateParams and PresentationParams for the Dialog Items. This is held in an unstructured array of bytes, as large as necessary to contain the data.

Note that this is a fully recursive structure. More than one level of child windows may be specified.

#### **4.1.8.3 Dialog Control Groups**

Within a Dialog Box, sets of controls can be aggregated into Groups, mainly for the purposes of easier interaction for the end-user.

Groups are really useful for Button controls, but could apply to other types of controls.

When a set of controls are formed into a group, the user can go from one to the next by pressing the arrow keys on the keyboard. The focus progresses round all the members of the group, but not members of any other group. Note that this implies that controls which themselves process the arrow keys, such as Edit Controls, cannot be formed into a group.

A group is established by setting the `WS_GROUP` style bit for the first control in the group. The other members of the group should follow the first in the enumeration order (ie follow in the template). The group is terminated by the next control which has the `WS_GROUP` style - this starts the next group. Logically, all the items in a Dialog Box are in one group or another.

The user can go from one group to the next by using the TAB keys. When the TAB key is pressed, the focus is moved to the next control in the enumeration order which has the `WS_TABSTOP` style bit set. Normally this would be the first member of a group.

Tab keys are the normal way to get from one Edit Control to the next and these controls should always have the `WS_TABSTOP` style. In fact, this is the default for Dialog Templates generated from text resource files by the resource compiler.

#### **4.1.8.4 Dialog Box Messages**

A dialog procedure, being a standard window procedure, receives any messages that are sent to the dialog box. This includes the standard window management messages such as `WM_DESTROY`, as well as control notifications such as `WM_CONTROL` and `WM_COMMAND`.

The dialog procedure may also receive WM\_SUBSTITUTEESTRING messages, if substitution strings are used in the dialog template. If sent, these messages are sent before an item is created. See "WinSubstituteStrings()".

Presentation Manager sends the following message after a dialog box is created, but before it is made visible:

---

## WM\_INITDIALOG

---

### Format

Message:	UINT	WM_INITDIALOG
lParam1:	HWND	hwndSetFocus
lParam2:	LPSTR	lpCreateParams
Returns:	BOOL	fFocusSet

### Description

This message is sent by WinLoadDlg or WinCreateDlg to a dialog procedure right after a dialog window is created, but before it is shown. This message is intended to allow the application to do run time initialization of the dialog box. Often it will be necessary to initialize text in static and edit controls and listboxes at this time, or to check buttons according to the current state of the application. hwndSetFocus is the window handle of the control that is going to receive the focus. The application may change this by calling WinSetFocus() with the window handle of another control in the dialog box and returning TRUE. It should otherwise return FALSE.

lpCreateParams is the lpCreateParams parameter passed to WinLoadDlg or WinCreateDlg.

**Notes** If any string substitutions were made by WinSubstituteString() when the dialog was created, WM\_SUBSTITUTEESTRING message may have been sent BEFORE the WM\_INITDIALOG message is sent.

When a dialog is created with WinCreateDlg() or WinLoadDlg(), a WM\_ADJUSTWINDOWRECT message is sent to each child window after the window is created. See the section dealing with the WM\_ADJUSTWINDOWRECT message for more detail.

#### 4.1.8.5 Dialog Styles

The following styles are defined especially for dialog boxes:

---

Dialog Window Styles  
Style

##### DS\_DLGFRAME

The dialog box is created with a special frame that identifies it as a dialog box. Most dialog windows are created with this style.

##### DS\_SCREENALIGN

The coordinates specifying the location of the dialog box are relative to the top left corner of the screen, rather than being relative to the owner window's origin.

##### DS\_MOUSEALIGN

The coordinates specifying the location of the dialog box are relative to the position of the mouse cursor at the time that the window was created. Presentation Manager will attempt to keep the dialog box on the screen, if possible.

#### 4.1.8.6 Dialog Box Routines

---

##### WinLoadDlg

---

Format

```
HWND WinLoadDlg(hab, hwndOwner, lpfnDlgProc,
                idModule, idDlg, lpCreateParams )
HWND      hwndOwner;
FARPROC   lpfnDlgProc;
UINT      idModule;
UINT      idDlg;
LPSTR     lpCreateParams;
HAB hab;
```

Description

This function creates a dialog window from a template in a resource. `hwndOwner` is the owner window. `lpfnDlgProc` is the dialog procedure, or NULL if it doesn't exist. `idModule`, returned by the `DOS DosLoadModule` call, is a resource handle or NULL for the application resource file. `idDlg` is the id of the dialog within the resource file. `lpCreateParams` points to data that is passed to the dialog procedure along with a `WM_INITDIALOG` message. This function returns the window handle of the new dialog box.



**Notes** WinLoadDlg returns immediately after creating the dialog box. A WM\_INITDIALOG message is sent to the dialog procedure before this function returns.

Also, since each of the controls will be created when this function is called, the dialog procedure may receive various control notifications.

A dialog window may be destroyed with WinDestroyWindow().

As windows are created from the template, strings in the template are processed with WinSubstituteString(). Any resultant WM\_SUBSTITUTESTRING messages are sent to the dialog procedure before WinLoadDlg() returns.

---

### WinCreateDlg

---

#### Format

```
HWND WinCreateDlg(hab, hwndOwner, lpfnDlgProc,
                  lpDlgTemplate, lpCreateParms)
HWND      hwndOwner;
FARPROC lpfnDlgProc;
DLGTEMPLATE FAR *lpDlgTemplate;
LPSTR     lpCreateParms;
HAB hab;
```

#### Description

This function is identical to WinLoadDlg() (above) except that it creates a dialog window from a template in memory rather than a resource file. lpDlgTemplate points to a data structure defined by DLGTEMPLATE. This function returns the window handle of the new dialog box.

---

### WinProcessDlg

---

#### Format

```
UINT WinProcessDlg(hwndDlg)
HWND hwndDlg;
```

#### Description

This function processes messages intended for a modal dialog. It does not return until the WinDismissDlg call is issued by the dialog procedure.

If the application's message queue is not empty, this function dispatches the messages from the

queue to the appropriate window procedure or to the dialog procedure, until either a message causes the dialog procedure to issue the `WinDismissDlg` call, or until all the messages from the application queue are processed.

If, or when, the application's message queue is empty, this function guarantees the visibility of the modal dialog identified by the `hwndDlg` parameter, i.e. shows the dialog window if it is hidden.

The function returns with the `wResult` parameter that was provided to the `WinDismissDlg` function, which has hidden the dialog window but has not destroyed it.

Character messages are processed by `WinProcessDlgMsg` to implement the standard user interface user interface. Messages are processed in this way ONLY if `WinDispatchMsg()` returns `FALSE` (i.e. the window with the focus did not process the message).

---

## WinDlgBox

---

### Format

```
UINT WinDlgBox(hab, hwndOwner, lpfnDlgProc,
               idModule, idDlg, lpCreateParams)
HWND      hwndOwner;
FARPROC   lpfnDlgProc;
UINT      idModule;
UINT      idDlg;
LPSTR     lpCreateParams;
HAB hab;
```

### Description

This function creates a modal dialog box. The parameters to this function are the same as the parameters to `WinLoadDlg()`. This function does not return until the dialog procedure calls `WinDismissDlg()` to destroy the dialog box and relinquish control. The value returned by this function is equal to the value of the `wResult` parameter passed to the `WinDismissDlg()` function.

```
This function is equivalent to:
hwndDlg = WinLoadDlg(...);
result = WinProcessDlg(hwndDlg);
WinDestroyWindow(hwndDlg);
return (result);
```

**Notes** Values returned by the application's dialog procedure are processed by Presentation Manager. They are not returned to the caller of WinDlgBox().

---

## WinDismissDlg

---

### Format

```
VOID FAR WinDismissDlg (hwndDlg, wResult)
HWND hwndDlg;
UINT wResult;
```

### Description

This function hides the dialog box window and sets a flag that causes the WinProcessDlg()/WinDlgBox() message processing loop to terminate. WinDismissDlg() is required to complete processing whenever the WinProcessDlg()/WinDlgBox() function is used to create a modal dialog box. This function is called from within a dialog procedure. wResult is the value which will be returned to the caller of WinProcessDlg()/WinDlgBox().

**Notes** If necessary, WinDismissDlg() can be called during the processing of the WM\_INITDIALOG message.

---

## WinSendDlgItemMsg

---

### Format

```
ULONG WinSendDlgItemMsg (hwndDlg, idItem, msg,
                          lParam1, lParam2)
HWND hwndDlg;
UINT idItem;
UINT msg;
ULONG lParam1;
ULONG lParam2;
```

### Description

This function is used to send a message to the specified dialog item identified by idItem in the dialog box specified by hwndDlg. msg, lParam1 and lParam2 are the SendMessage parameters. WinSendDlgItemMsg does not return until the message has been processed. This function returns the value returned by the dialog item's window function.

Notes     This function is equivalent to:

```
WinSendMessage (WinWindowFromID (hwndDlg, idItem), msg, lParam)
```

---

### WinSetDlgItemInt

---

#### Format

```
BOOL WinSetDlgItemInt (hwndDlg, idItem, wValue, fSigned)
HWND hwndDlg;
UINT idItem;
UINT wValue;
BOOL fSigned;
```

#### Description

This function sets the text of the item with `idItem` in the specified to the string representation of the integer `wValue`. If `fSigned` is `TRUE`, `wValue` is a signed integer; otherwise, it is an unsigned integer.

Notes     The string is always produced as an ASCII string.

---

### WinQueryDlgItemInt

---

#### Format

```
INT WinQueryDlgItemInt (hwndDlg, idDlgItem,
                        lpfSuccess, fSigned)
HWND hwndDlg;
UINT idDlgItem;
LPSTR lpfSuccess;
BOOL fSigned;
```

#### Description

This function translates the text of a dialog item into an integer value. `hwndDlg` is the dialog window handle, `idDlgItem` is the id of the item to translate. If `fSigned` is `TRUE`, `WinGetDlgItemInt` checks for a minus sign before translating the number. If `lpfSuccess` is non-zero, it points to a boolean variable. `WinQueryDlgItemInt` sets this boolean variable to `TRUE` if it succeeds in translating the text without any errors. Otherwise, it sets this variable to `FALSE`. This function returns the translated integer value.

Notes     The dialog item string is assumed to be an ASCII string.

---

**WinMapDlgPoints**

---

**Format**

```
VOID WinMapDlgPoints (hwndDlg, lprgpt, cpt,  
                     fCalcWindowCoords)  
HWND hwndDlg;  
POINT FAR *lprgpt;  
INT cpt;  
BOOL fCalcWindowCoords;
```

**Description**

This function maps points from dialog coordinates to window coordinates, or from window coordinates to dialog coordinates, depending on the state of `fCalcWindowCoords`.

`lprgpt` is a far pointer to an array of `POINT` structures, and `cpt` is the number of points in this array. `hwndDlg` is the dialog box handle that the coordinates will be mapped to or from.

If `fCalcWindowCoords` is `TRUE`, the points in the array are assumed to be in dialog coordinates. They are mapped to window coordinates relative to `hwndDlg`.

If `fCalcWindowCoords` is `FALSE`, the points in the array are assumed to be in window coordinates relative to `hwndDlg`. They are mapped to dialog coordinates.

---

**WinProcessDlgMsg**

---

**Format**

```
BOOL WinProcessDlgMsg (hwndDlg, lpqmsg)  
HWND hwndDlg;  
LPQMSG lpqmsg;
```

**Description**

This function determines whether the message specified by `lpqmsg` is intended for the modeless dialog box specified by `hwndDlg`. If so, the message is sent to the dialog window and this function returns `TRUE`. Otherwise, this function returns `FALSE`. This function is typically called from within an application's main loop to allow a modeless dialog box to process the appropriate messages, while passing other messages on to its window proc.

Notes     if WinProcessDlgMsg returns TRUE, the message has already been processed and should NOT be passed on to WinDispatchMsg().

```
while (WinGetMsg((LPQMSG)&qmsg, (HWND)NULL, 0, 0)) {  
    if (!WinProcessDlgMsg(hwndModelessDlg, (LPQMSG)&qmsg)) {  
        WinDispatchMsg(LPMSG)&msg;  
    }  
}  
exit(qmsg.lParam1);
```

---

## WinEnumDlgItem

---

### Format

```
HWND WinEnumDlgItem(hwndDlg, hwnd, code, fLock)  
HWND     hwndDlg;  
HWND     hwnd;  
UINT     code;  
BOOL     fLock;
```

### Description

This function is used to obtain the next or previous window handle of a dialog item with either the WS\_TABSTOP style bit set, or in the same "group" as the starting hwnd.

This function always returns a window handle that is an immediate child of hwndDlg, even if hwnd is not an immediate child.

If fLock is TRUE, return the window handle locked.

Wraps around to beginning when obtaining next at end of list. Wraps around to end when obtaining prev at beginning of list.

Available code values are shown below. Note that only one of these values may be used.

---

### WinEnumDlgItem() Codes

Returns:

#### EDL\_PREVTABITEM

Previous item with style  
WS\_TABSTOP; wraps around to end

#### EDL\_NEXTTABITEM

Next item with style WS\_TABSTOP;  
wraps around to beginning

#### EDL\_FIRSTTABITEM

First item in dialog with style  
WS\_TABSTOP; hwnd is ignored

**EDL\_LASTTABITEM**

Last item in dialog with style  
WS\_TABSTOP; hwnd is ignored

**EDL\_PREVGROUITEM**

Previous item in the same group; wraps  
around to end

**EDL\_NEXTGROUITEM**

Next item in the same group; wraps  
around to beginning

**EDL\_FIRSTGROUITEM**

First item in the current group

**EDL\_LASTGROUITEM**

Last item in the current group

### 4.1.9 Message Boxes

A message box is a special predefined dialog window that any application can use to display messages and get simple input from the user. A message box contains a specified caption and message and up to four pushbuttons. The message box is displayed in the center of the screen. Message boxes can also contain any one of a predefined set of icons.

The message box may be "Application Modal" or "System Modal". Application modal boxes do not allow the user to activate any other window in belonging to the same application before responding to the message box, while System Modal message boxes disable the entire system. System modal message boxes are used to notify the user of serious, potentially damaging, errors that require immediate attention (for example, running out of memory).

#### 4.1.9.1 Message Box Functions

---

**WinMessageBox**

---

**Format**

```
UINT FAR WinMessageBox(hab, hwndOwner, idHelp,  
                        lpszText, lpszCaption, rgfStyle)  
HAB hab;  
HWND hwndOwner;  
UINT idHelp;  
LPSTR lpszText;  
LPSTR lpszCaption;  
UINT rgfStyle;
```

### Description

This function creates and displays a message box. `hwndOwner` specifies the owner of the message box and `lpszText` and `lpszCaption` specify the text content and the caption content of the message box respectively. If `lpszCaption` is `NULL`, the default caption "Error" is displayed. The Owner window is activated when the `WinMessageBox` returns.

`rgfStyle` is a bit array specifying the contents and function of the message box. Any of the following combinations can be used:

If multiple lines are required in the text of the message, carriage return characters may be inserted in `*lpszText`.

The window ID of the message box is set to `idHelp`. This value is passed to the `HK_HELP` hook if the `WM_HELP` message is received by the message box. See the description of the `HK_HELP` hook for more detail.

---

### `WinMessageBox()` Styles Meaning

#### `MB_OK`

Message box contains an Ok push button.

#### `MB_OKCANCEL`

Message box contains Ok and Cancel push buttons.

#### `MB_RETRYCANCEL`

Message box contains Retry and Cancel push buttons.

#### `MB_ABORTRETRYIGNORE`

Message box contains three push buttons: Abort, Retry, and Ignore.

#### `MB_YESNO`

Message box contains two push buttons: Yes and No .

#### `MB_YESNOCANCEL`

Message box contains three push buttons: Yes, No, and Cancel.

#### `MB_HELP`

Message box contains a help pushbutton.



**MB\_ICONHAND**

A hand icon appears in the message box.

**MB\_ICONQUESTION**

A question mark icon appears in the message box.

**MB\_ICONEXCLAMATION**

An exclamation point icon appears in the message box.

**MB\_ICONASTERISK**

An asterisk icon appears in the message box.

**MB\_DEFBUTTON1**

First button is the default (the first button is always the default unless **MB\_DEFBUTTON2** or **MB\_DEFBUTTON3** is specified).

**MB\_DEFBUTTON2**

Second button is the default.

**MB\_DEFBUTTON3**

Third button is the default.

**MB\_APPLMODAL**

The message box is application modal.

**MB\_SYSTEMMODAL**

The message box is system modal.

**Return Value**

Message box returns one of the following values depending on the user's response:

---

**Message Box Return Values**

**Return Value**

**IDOK** OK button was pressed

**IDCANCEL** Cancel button was pressed

**IDABORT** Abort button was pressed

**IDRETRY** Retry button was pressed

**IDIGNORE** Ignore button was pressed

**IDYES** Yes button was pressed

IDNO No button was pressed

If a message box has a Cancel button, the IDCANCEL value will be returned if either the Escape or Cancel key is pressed. If the message box has no Cancel button, pressing the Escape key has no effect.

**Notes** When a system modal message box is created to indicate that the user is low on memory, the strings passed as the `lpszText` and `lpszCaption` parameters should not be taken from a resource file, since an attempt to load a resource file may fail. The message box can safely use the hand icon (`MB_HANDICON`), since this icon is always resident and does not require disk access.

When the keyboard interface is used to enumerate windows, the message box and its parent are considered to be next to each other.

If a message box is created while a dialog box is present, use the handle of the dialog box as the `hwndOwner` parameter.

---

## WinAlarm

---

### Format

```
BOOL FAR WinAlarm(hab, rgfType)
HAB hab;
UINT rgfType;
```

### Description

This function generates a beep at the system speaker. This function returns TRUE if a beep is actually generated. `rgfType` may be one of the following:

```
WA_WARNING
WA_NOTE
WA_ERROR
```

---

## WinSubstituteStrings

---

### Format

```
INT FAR WinSubstituteStrings(hwnd, lpszSrc,
                             lpszDst, cchDstMax);
HWND hwnd;
LPSTR lpszSrc;
LPSTR lpszDst;
```

INT cchDstMax;

#### Description

This function copies a null terminated string from \*lpszSrc to \*lpszDst, performing substitution of certain characters with application-supplied strings as the string is copied.

If a string of the form "%<digit>" is encountered, where <digit> is a digit from "0" to "9", a WM\_SUBSTITUTESTRING message is sent to the specified window. This message returns a far pointer to a null terminated string that is copied to the destination, replacing the "%<digit>" string.

If two "%" characters are encountered, only one is copied. If the character following the "%" is not a digit or "%", both the "%" and the character following are copied.

This function returns the length of lpszDst, not including the null termination character. If the destination string would be longer than cchDstMax characters, it is truncated at cchDstMax - 1 characters; the string is always null terminated.

---

### WM\_SUBSTITUTESTRING

---

#### Format

WM\_SUBSTITUTESTRING  
LOUINT(1Param1): INT iString;  
1Param2: OL;  
Returns: LPSTR lpszSubstituteString;

#### Description

This message is sent by WinSubstituteStrings() when a substitution string of the form "%<digit>" is encountered, where <digit> is a character between "0" and "9". iString is a number between 0 and 9 corresponding to <digit>, identifying which substitution is being made.

This message returns a far pointer to a NULL terminated string, which will be substituted for the "%<digit>" string.

See WinSubstituteStrings() above.

#### Example

Here is an example of how WinSubstituteStrings() and the WM\_SUBSTITUTESTRING message can

be used:

The call:

```
cch = WinSubstituteStrings(hwnd,  
                           "Do you want to %0 the file %1?",  
                           lpchBuffer, cchBufferMax);
```

with the following case in the window procedure of  
hwnd:

```
case WM_SUBSTITUTESTRING:  
    switch (lParam1) {  
        case 0:  
            return((LPSTR) "delete");  
            break;  
        case 1:  
            return((LPSTR) "EXAMPLE.EXE");  
            break;  
    }  
    break;
```

produces the string:

```
Do you want to delete the file EXAMPLE.EXE?
```

#### 4.1.10 Control Windows

"Control windows", or just "controls", are predefined classes for child windows that any application can use for input and output. Controls are usually part of a dialog box and are loaded from a dialog template, but they can also be created by the application by calling `WinCreateWindow()` with the appropriate control class. The following control classes have been predefined:

---

Standard Control Classes
Class Name

<b>WC_BUTTON</b>
------------------

These controls consist of buttons and boxes that the user can select by clicking the mouse or using the keyboard.
---

<b>WC_EDIT</b>
----------------

These controls consist of a single line of text that the user can edit.
---

<b>WC_STATIC</b>
------------------

These controls are simple display items such as icons and text that do not respond to keyboard or mouse events. They are used primarily inside dialog boxes.
--

<b>WC_LISTBOX</b>
-------------------

These controls present a list of text items from which the user can make selections.
--

**WC\_MENU**

These controls also present a list of items. The items may be text or bitmaps, displayed horizontally as "Action Bars" or vertically as "Popup Menus". Menus are usually used to provide a command interface to applications.

**WC\_SCROLLBAR**

These controls consist of window scroll bars that allow the user to make a request to scroll the contents of an associated window.

**WC\_MINMAXBOX**

Minimize/Maximize pushbuttons

**WC\_SIZEBORDER**

Window sizing border

**WC\_TITLEBAR**

This control displays the window title or caption and allows the user to move its owner by dragging the control.

**4.1.10.1 Common Features**

Control windows are just like any other window, so all of the standard window management functions such as `WinSetWindowText()` and `WinShowWindow()` can be used. In addition, they handle the standard window messages such as `WM_QUERYWINDOWPARAMS` and `WM_ENABLE`. See the section on "Window Management".

Although there are differences among controls of different classes, they all share the following common features.

**4.1.10.1.1 Synchronous Painting**

Controls are typically painted synchronously. For more information, see "Synchronous Window Updating"

**4.1.10.1.2 Control Identification**

Like all windows, a control has a unique ID value that identifies it. This value is determined by the application when the control is created. See "Window Attributes". This id is used by the control when it notifies its owner.

#### 4.1.10.1.3 Control States

Controls can be enabled and disabled, and hidden and shown with `WinEnableWindow()` and `WinShowWindow()`. A disabled control is usually halftoned to indicate to the user that it is disabled. Some controls can be hilited by sending them a `SETHILITE` message. For example a titlebar control is hilited by sending it a `TBM_SETHILITE` message.

#### 4.1.10.1.4 Control Messages

Typically, control windows each have a certain set of messages that are used to interact with the control. For instance, menu controls will respond to window messages to add, insert, and delete menu items. Each control has its own set of messages.

#### 4.1.10.1.5 Control data

Some controls respond to their control data. They are initially passed the control data via the `WM_CREATE` message. They have to respond to changes to the control data that is sent to them by `WM_SETWINDOWPARAMS` messages, and return their control data in response to `WM_QUERYWINDOWPARAMS` messages.

The following defines the control data for the standard window classes.

```
WC_BUTTON
    typedef struct {
        UINT wCheckState;
        UINT wHiliteState;
    } BTNCDATA;
```

`wChecked` is a uint that indicates the current button check state, i.e., the same as the value returned by/passed to the `BM_QUERYCHECK/BM_SETCHECK` message.

`wHiliteState` is a uint that indicates the current button hilite state, as returned by/passed to `BM_QUERY/SETHILITE`.

```
WC_MENU
```

`ControlData` points to a menu template structure. The menu control data may not be queried or set after the window is created. The `ControlData` field is interpreted by the menu control only when the control is created.

```
WC_EDIT
    typedef struct {
        UINT cchEditLimit;
    } EDITCDATA;
```

`cchEditLimit` contains the maximum number of characters that may be entered into the edit control.

```
WC_SCROLLBAR
    typedef struct {
        UINT posFirst;
        UINT posLast;
        UINT posThumb;
    } SBCDATA;
```

`posFirst` and `posLast` reflect the scroll range, and `posThumb` is the current thumb position.

`WC_FRAME`

No control data.

`WC_DIALOG`

No control data.

`WC_TITLEBAR`

No control data.

`WC_SIZE`

No control data.

`WC_STATIC`

No control data.

`WC_LISTBOX`

No control data.

#### *4.1.10.1.6 Control Owner Notifications*

When interesting things happen with a control, the owner window of the control is notified with a window message. The message sent, and whether the message is posted or sent depends on the control. For example, menu controls post `WM_COMMAND` messages to their owner when an item is selected. Many controls send notifications via the `WM_CONTROL` message.

#### 4.1.10.1.7 *Dialog Codes*

When processing user input, the dialog manager makes some assumptions about how certain controls operate. For example, when an edit control is given the focus, its entire text is selected.

A control's "dialog code" contains flags that govern what assumptions the dialog manager makes about a control. The dialog manager sends the `WM_QUERYDLGCODE` message to get a control's dialog codes.

#### 4.1.10.1.8 *Control Heap*

Most controls require memory for the storage of data associated with that control, such as the strings that make up the items in a menu control, or the text buffer associated with an edit control.

This data is normally allocated in a heap created with the `WinCreateHeap()` function. Controls typically send a `WM_CONTROLHEAP` message to their owner in order to obtain the heap handle to allocate in. Some controls do not require a heap handle; others create their own heap handle.

For each thread with a message queue, there is an associated heap handle that can be used by controls (or applications) for the purpose of allocating app-specific data. This heap handle is returned when `WM_CONTROLHEAP` is processed by `WinDefWindowProc()`. See the "Local Memory Manager".

### 4.1.10.2 **Standard Control Messages**

The following messages can be sent to all controls:

---

#### `WM_ADJUSTWINDOWRECT`

---

##### Format

```
WM_ADJUSTWINDOWRECT
lParam1:    LPSWP    lpswp
lParam2:    0
Returns:    BOOL      fAdjusted
```

##### Description

This message is sent to controls by `WinSetWindowPos()` to allow the control window to adjust its new position or size whenever it is about to be moved. `lpswp` points to an `SWP` structure which has been filled in by `SetWindowPos()` with the proposed move/size data. The control may adjust



this new position by changing the contents of the SWP structure. It can change the x/y fields to adjust its new position; it can change the cx/cy fields to adjust its new size, or it change the `hwndInsertBehind` field to adjust its new z-order.

Frame controls can respond to this message to reposition themselves or resize themselves in the window frame. Menu controls respond to this message by resizing both their height and width to fit the current content of the menu.

The return value is `TRUE` if any change has been made to the SWP structure, `FALSE` otherwise.

When a dialog is created with `WinCreateDlg()` or `WinLoadDlg()`, a `WM_ADJUSTWINDOWRECT` message is sent to each child window after the window is created, with a pointer to an SWP structure containing `rgf == SWP_SIZE | SWP_MOVE`, and the x, y, cx, cy fields initialized to the window's current size and position. The message allows the control to adjust its size or position, usually to compensate for its border and/or margin.

The `WM_ADJUSTWINDOWRECT` is only processed by listboxes and edit controls:

---

#### Listboxes:

Listboxes automatically outsets its border in addition to changing its height to accomodate an exact number of items. This means that the "x, y, cx, cy" fields in the .rc file specify the working area of the listbox. The border will be drawn outside this area.

#### Edit controls:

The edit control, if `ES_MARGIN` is specified, outsets its margin. This means that in the .rc file, the numbers specified as the x, y position of an edit control are taken to be the position where the first character of text is drawn, not where the lower left corner of the surrounding box is drawn. Similarly, the height and width parameters apply to the editable area of the control, thus not including the margin.

---

## WM\_QUERYDLGCODE

---

### Format

```
WM_QUERYDLGCODE
lParam1:      LPQMSG *lpQmsg;
lParam2:      0
Returns:      ULONG lDialogCode;
```

### Description

When processing user input, the dialog manager makes some assumptions about how certain controls operate. The dialog manager sends the WM\_QUERYDLGCODE message to obtain a code that govern what assumptions can be made.

This message is sent by the dialog manager to identify the type of control to determine what kinds of messages the control will understand, and also to determine whether an input message may be processed by the dialog manager or passed down to the control.

The following information flags (which can be OR'ed together) are returned. There are two groups of flags: those that identify the type of control and the messages that can be sent to it, and those that control how the dialog manager handles user input:

---

### Dialog Codes

#### Dialog Code

#### DLGC\_EDIT

Identifies an edit control. Assumed to understand the EM\_SETSEL message.

#### DLGC\_BUTTON

Identifies a button item. Assumed to understand the BM\_CLICK message.

#### DLGC\_CHECKBOX

Identifies a checkbox item. Used with the DLGC\_BUTTON code.

#### DLGC\_RADIOBUTTON

Identifies a radio button control. Used with the DLGC\_BUTTON code.

#### DLGC\_STATIC

Identifies a static control. Static controls are not included in arrow key enumeration.

**DLGC\_DEFPUSHBUTTON**

Identifies a Default pushbutton control

**DLGC\_UNDEFPUSHBUTTON**

Identifies a Non-default pushbutton

**4.1.10.3 Owner Notification Messages**

Controls are useful because they notify their owners when interesting events take place. A control notifies its owner by sending a **WM\_CONTROL** message or by posting a **WM\_COMMAND** or **WM\_HELP** message.

---

**WM\_CONTROL**

---

## Format

```
WM_CONTROL
LOUINT(lParam1): UINT      id
HIUINT(lParam1): UINT      wNotifyCode
lParam2:         ULONG      lControlSpec
```

## Description

A **WM\_CONTROL** message is always sent (via **WinSendMsg()**.)

**LOUINT(lParam1)** contains the window ID of the control, which is the ID parameter of **WinCreateWindow()** or in a dialog template.

**HIUINT(lParam1)** contains what is known as a "notification code", which is a value that indicates why the notification is taking place. The values of the notification codes sent by each control depends on the control class.

**lParam2** contains control-specific information, which is often simply the window handle of the control.

---

**WM\_COMMAND**

---

## Format

```
WM_COMMAND
LOUINT(lParam1): UINT      id
LOUINT(lParam2): UINT      wSource;
HIUINT(lParam2): BOOL      fMouse;
```

**Description**

Menu controls and pushbuttons commonly post WM\_COMMAND messages to the message queue.

lParam1 contains the window ID of the control, which is the ID parameter of WinCreateWindow() or in a dialog template.

LOUINT(lParam2) contains information that indicates the source of the WM\_COMMAND message:

---

WM\_COMMAND Source Codes  
Source Code

CMDSRC\_PUSHBUTTON  
Posted by a pushbutton control

CMDSRC\_MENU  
Posted by a menu control

CMDSRC\_ACCELERATOR  
Posted by WinTranslateAccelerator().

CMDSRC\_OTHER  
Other source

HIUINT(lParam2) contains TRUE if the WM\_COMMAND message was posted as a result of a mouse operation, FALSE if a keyboard operation.

---

**WM\_HELP**

---

**Format**

```
WM_COMMAND
LOUINT(lParam1): UINT    id
LOUINT(lParam2): UINT    wSource;
HIUINT(lParam2): BOOL    fMouse;
```

**Description**

Menu controls and pushbuttons post WM\_HELP messages to the message queue when they have appropriate style. The implication is that the application should respond to the selection of the item by displaying help information. Otherwise, the message is identical to a WM\_COMMAND message.

lParam1 contains the window ID of the control, which is the ID parameter of WinCreateWindow() or in a dialog template.

LOUINT(lParam2) contains information that

indicates the source of the WM\_HELP message:

---

#### WM\_HELP Source Codes

Source Code

#### CMDSRC\_PUSHBUTTON

Posted by a pushbutton control

#### CMDSRC\_MENU

Posted by a menu control

#### CMDSRC\_ACCELERATOR

Posted by WinTranslateAccelerator().

#### CMDSRC\_OTHER

Other source

HIUINT(lParam2) contains TRUE if the WM\_HELP message was posted as a result of a mouse operation, FALSE if a keyboard operation.

---

### WM\_CONTROLCOURSEUR

---

#### Format

LOUINT(lParam1):	UINT	idCtl;
lParam2:	HWND	hwndCtl;
Returns:	BOOL	fProcessed;

#### Description

Sent to a control's owner window when the mouse cursor moves over the control window, allowing the owner to set the mouse cursor. If FALSE is returned, the control will set the cursor as normal. If TRUE is returned, the control will not set the cursor.

---

### WM\_CONTROLHEAP

---

#### Format

WM_CONTROLHEAP		
LOUINT(lParam1):	UINT	id
lParam2:	HWND	hwndCtl
Returns:	HANDLE	hHeap

#### Description

This message is sent by some controls to their owner in order to obtain the heap handle to use for allocating control-specific data. This message should be handled by returning a heap handle to

use, which has been created with `WinCreateHeap()`. `id` is the control id, and `hwndCtl` is the window handle of the control.

`WinDefWindowProc()` handles this message by returning the queue heap handle.

See the "Local Memory Manager".

### 4.1.11 Static Controls

Static controls are simple text fields, boxes and rectangles that can be used to label, box or separate other controls. Static controls do not accept input, nor do they notify their owner.

A static control can be disabled, even though it cannot receive input. A disabled static control is halftoned. This can be used to indicate that the text is not applicable for some reason.

Static text items may contain mnemonic characters, which are emboldened. If the text of a static control contains a mnemonic character, the text is displayed with the appropriate mnemonic highlighting.

Static controls can never receive the focus. Whenever a static control receives a `WM_SETFOCUS` message, or whenever the user clicks on a static control, that static control advances the focus to the next sibling that is not a static control. (Inside dialog boxes, the next non-static control will receive the focus.) If the static control has no such siblings, it sets the focus to its owner.

#### 4.1.11.1 Static Control Styles

There are the following static control styles:

---

Static Control Styles
Style

<b>SS_ TEXT</b>
-----------------

Creates a text field. The text is formatted before it is displayed. Words that would extend past the end of a line are automatically wrapped to the beginning of the next line. Variations of the <b>SS_ TEXT</b> style are created by ORing various <code>WinDrawText()</code> flags into the style.
---

The possible flags are:
-------------------------

---

<b>DT_ LEFT</b>
-----------------

Left Justified text
---------------------

**DT\_CENTER**

Centered text

**DT\_RIGHT**

Right justified text

**DT\_TOP**

Text is aligned to top of window

**DT\_VCENTER**

Text is aligned vertically in center of window

**DT\_BOTTOM**

Text is aligned to bottom of window

**DT\_WORDBREAK**

When specified with **DT\_TOP**, this allows for multi-line static text controls, with word wrapping at the ends of lines.

See the `WinDrawText()` documentation for a more complete description of these flags.

**SS\_GROUPBOX**

A groupbox static control is a rectangle that has an identifying text string in its upper left corner. Group boxes are used to collect a group of radio buttons or other controls in a single unit.

**SS\_ICON**

Draws an icon. The text of the static control is a string which is used to derive the resource ID from which the icon is loaded. The format of the string is:

1. First byte 0xff, 2nd byte is the low byte of the resource ID, 3rd byte is the hi byte of the resource ID.
2. First character is "#", subsequent characters make up the decimal text representation of the resource ID.

If the string is empty or does not follow the format above, no resource is loaded.

The resource is assumed to reside in the resource file of the current process.

**SS\_BITMAP**

Draws a bitmap. The text of the static control names the bitmap resource, as for **SS\_ICON** above.

**SS\_FGNDRECT**

Creates a foreground color filled rectangle.

**SS\_BKGDRECT**

Creates a background color filled rectangle.

#### **SS\_FGNDFRAME**

Creates a box with a foreground color frame.

#### **SS\_BKGNDFRAME**

Creates a box with a background color frame.

### **4.1.11.2 Static Notification Codes**

Static items do not generate notification messages.

## **4.1.12 Button Controls**

A button control is a small rectangular child window that represents a button that the user can turn on or off by clicking it with the mouse. Button controls can be used alone or in groups, and can either be labeled or appear without text. Button controls typically change appearance when the user clicks them.

Buttons can be disabled to prevent them from responding when the user clicks on them. Disabled buttons are halftoned.

The class name to use when creating button controls is **WC\_BUTTON**.

### **4.1.12.1 Button Control Styles**

Button controls have the following window styles:

---

#### **Button Control Styles**

Style

##### **BS\_PUSHBUTTON**

A pushbutton is a box that contains a string. When you push a button, the parent window is notified. You push a button by clicking the mouse on it or pressing the **SPACEBAR** when the button is active.

##### **BS\_DEFPUSHBUTTON**

A defpushbutton (or default pushbutton) is a pushbutton with a thick border box. It has the same properties as a pushbutton. In addition, the user may press a defpushbutton by pressing the **RETURN** key.

##### **BS\_CHECKBOX**

A checkbox is a little square with a character string to the right. If it is checked, a small black box appears inside the little square. When you click the box or string, the checkbox changes state and the parent window is notified. You click the box by clicking on it with the mouse or pressing the



SPACEBAR when it is active.

**BS\_AUTOCHECKBOX**

An automatic check box automatically toggles its state whenever the user clicks on it.

**BS\_RADIOBUTTON**

A radio button is similar to a check box, but is typically used in groups in which only one button at a time is checked. When a radio button is clicked or a cursor key is pressed to move within the group, it notifies its owner window. It is then up to the owner window to check the clicked radio button and uncheck all the rest, if necessary.

**BS\_AUTORADIOBUTTON**

When clicked, an automatic radio button automatically checks itself and unchecks all other radio buttons in the same group.

**BS\_3STATE**

A 3-state check box is identical to a check box control except that its check box can be halftoned as well as the box being checked or unchecked.

**BS\_AUTO3STATE**

An automatic 3-state check box automatically toggles its state when the user clicks on it.

**BS\_USERBUTTON**

This is a application-definable button. The owner window of this style control will receive four additional button messages: BN\_HILITE, BN\_UNHILITE, BN\_DISABLE, and BN\_PAINT (documented below).

**BS\_HELP**

The button posts a WM\_HELP message rather than a WM\_COMMAND or WM\_SYSCOMMAND message. In addition, BS\_HELP buttons do not set the focus to themselves when clicked with the mouse. This style is useful for producing a button which can be used for selecting help information.

#### 4.1.12.2 Button Control Messages

Button controls process the following message:

---

**BM\_CLICK**

Format

```
BM_CLICK
LOUINT(1Param1):    BOOL fUp;
```

lParam2: OL  
Returns: OL

#### Description

A button control sent a `BM_CLICK` message will take whatever action that would occur if the button was clicked with the mouse or space bar. If `fUp` is `TRUE`, then the default upclick action is taken; otherwise the default downclick action is taken.

---

### `BM_QUERYCHECKINDEX`

---

#### Format

`BM_QUERYCHECKINDEX`  
lParam1: OL  
lParam2: OL  
Returns: `INT` `iRadioButton`

#### Description

Returns the 0-based index of the checked radio button in the same group as the window that sent this message, or -1 if none are checked or if the button is not a valid radio button.

### 4.1.12.3 Button Notification Codes

#### 4.1.12.3.1 Pushbutton Notification Codes

When a pushbutton is clicked, a `WM_COMMAND` message is posted to the owner window. The following information is included in `lParam1` and `lParam2` of the `WM_COMMAND` message:

---

#### Parameter

##### Contents

#### `lParam1`

Contains the pushbutton control ID

#### `LOUINT(lParam2)`

Contains the value `CMDSRC_PUSHBUTTON`

#### `HIUINT(lParam2)`

Contains `TRUE` if the command was posted as a result of a mouse operation, and `FALSE` if a keyboard operation.

See "Owner Notification Messages" for more information.

#### *4.1.12.3.2 Radio Button and Checkbox Notification Codes*

Radio buttons and check boxes notify their owners with the WM\_CONTROL message.

---

### WM\_CONTROL: Button Notification

---

#### Format

```
WM_CONTROL
LOUINT(lParam1):  UINT idCtl;
HIUINT(lParam1):  UINT wNotifyCode;
lParam2:          HWND hwndCtl;
Returns:          OL;
```

#### Description

This message is sent to the button control owner when a radio button or checkbox is clicked with the mouse, or when it receives a BM\_CLICK message, or for BS\_USERBUTTON controls.

LOUINT(lParam1) contains the window ID of the button control. HIUINT(lParam1) contains a code that indicates the reason for the notification:

---

#### Button Control Notification Codes

Code
------

BN_CLICKED
------------

The button has been clicked with the mouse or the SPACEBAR is pressed when the button has the focus.
--

BN_DBLCLICKED
---------------

The button has been doubleclicked with the mouse.
---

BN_PAINT
----------

The button needs to be painted.
---------------------------------

BN_HILITE
-----------

The hilited state of the button needs to be painted.
--

BN_UNHILITE
-------------

The button should be unhilited.
---------------------------------

BN_DISABLED
-------------

The button should be painted in a disabled state.
---

#### 4.1.12.4 Button Control State Messages

Button controls can be hilited or unhilited. A hilited button is either inverted or emboldened. Radio buttons and checkboxes can be checked or unchecked. Three state buttons can be checked, unchecked or half-toned. `WinIsWindowEnabled()` may be used to determine whether or not a button is enabled or disabled.

The button controls may also be modified by the `WM_SET/QUERYWINDOWPARAMS` messages.

---

##### BM\_SETHILITE

---

###### Format

```
BM_SETHILITE
LOUINT(1Param1):    BOOL fHilite;
1Param2:             OL
Returns:             INT fHiliteOld;
```

###### Description

This message is used to cause the button control to be displayed in either the hilited or unhilited states.

Returns the previous hilite state of the button control.

---

##### BM\_QUERYHILITE

---

###### Format

```
BM_QUERYHILITE
1Param1:    O
1Param2:    OL;
Returns:    BOOL fHilite;
```

###### Description

This message returns the current hilite state of the button control. Returns `TRUE` if the button is hilited, `FALSE` otherwise.

---

##### BM\_SETCHECK

---

###### Format

```
BM_SETCHECK
LOUINT(1Param1):    wCheck
1Param2:             OL;
Returns:             wOldCheck;
```

**Description**

This message sets the current check state of the button control. Possible state values for a checkbox or radio button are TRUE (checked) and FALSE (unchecked). For three- state checkboxes, the possible state values are 0 (unchecked), 1 (checked), and 2 (indeterminate). Returns the previous check state.

---

**BM\_QUERYCHECK**

---

**Format**

```
BM_QUERYCHECK
lParam1:      0
lParam2:      OL;
Returns:      wCheck;
```

**Description**

This message returns the current check state of the button control. Returns current check state for the button control. Possible state values for a checkbox or radio button are TRUE (checked) and FALSE (unchecked). For three- state checkboxes, the possible state values are 0 (unchecked), 1 (checked), and 2 (indeterminate).

### 4.1.13 Edit Controls

An edit control is a rectangular window which displays a single line of text that the user can edit. When it has the focus, it displays a flashing caret to mark the current insertion point. It also allows the user to select parts of the text by clicking and dragging the mouse or by using the keyboard interface. Selected text can be cut or copied to the clipboard. Text from the clipboard can be inserted by pasting from the clipboard. Text which is pasted or entered from the keyboard either replaces the current selection or is inserted at the insertion point.

When an edit control is created, an initial buffer of 32 characters is allocated for it. The user can change this with the EM\_SETTEXTLIMIT message. The initial contents of the edit control can be set with the LPCREATESTRUCT parameter passed with the WM\_CREATE message.

If the user attempts to enter more text in an edit control than specified by the text limit set by the EM\_SETTEXTLIMIT message, the edit control indicates the error by beeping and does not accept the characters.

By default, edit controls use the current SYSTEM font to display characters, but the application can direct the control to use another font.

The class name to use when creating edit controls is WC\_EDIT.

#### 4.1.13.1 Edit Control Styles

There are the following edit control styles:

---

##### Edit Control Styles

###### Style

##### ES\_LEFT

The text in the control is left-justified. This is the default style if neither ES\_RIGHT or ES\_CENTER are specified.

##### ES\_RIGHT

The text in the control is right-justified.

##### ES\_CENTER

The text in the control is centered.

##### ES\_AUTOSCROLL

Whenever the user tries to move off the end of a line, the control automatically scrolls one third the width of the window in the appropriate direction.

##### ES\_MARGIN

This style can be used to cause a frame to be painted around the control, with a margin around the editable text. The margin is 1/2 character width wide, and 1/4 character height high.

If the user attempts to enter more text in an edit control than specified by the text limit set by the EM\_SETTEXTLIMIT message, the edit control indicates the error by beeping and does not accept the characters.

#### 4.1.13.2 Edit Control Keys

Edit controls allow the user to perform the following operations. The actual keys used to accomplish these functions are defined by the user interface.

1. Moves caret one character left
2. Moves caret one character right
3. Extends selection one character left

4. Extends selection one character right
5. Moves caret one word right
6. Moves caret one word left
7. Moves caret to beginning of line
8. Moves caret to the end of the line
9. Deletes the character to the left of the caret and resets the current selection
10. Deletes character to the right of caret and resets current selection
11. Cuts current selection to clipboard
12. Deletes current selection, but does not copy it to the clipboard
13. Replaces current selection with text contents of the clipboard

#### **4.1.13.3 Edit Control Notification Messages**

Edit controls send the following notification codes to their owner. For a general description of notification codes, see the section on Control Windows.

---

Edit Control Notification Codes
Code

EN_SETFOCUS
-------------

The edit control received the focus.
--------------------------------------

EN_KILLFOCUS
--------------

The edit control lost the focus.
----------------------------------

EN_CHANGE
-----------

The content of the edit control has changed, and the change has been displayed on the screen.
---

EN_SCROLL
-----------

The edit control display is about to scroll horizontally. This can happen the application has sent a scroll message to the edit control, or because the content of the edit control has changed, or the caret has moved, and the edit control must scroll to show current caret position.
---

#### 4.1.13.4 Edit Control Messages

An application can send or post the following messages to an edit control. (See the general control message section for a description of messages that can be sent to all controls.) An edit control also responds to the WM\_SET/QUERYWINDOWPARAMS messages.

---

##### EM\_GETCHANGED

---

###### Format

```
EM_GETCHANGED
lParam1:      0
lParam2:      0
Returns:      BOOL fChanged
```

###### Description

Returns the change ("dirty") state of the edit control. fChange is TRUE if the text in the edit control has changed since the last time it received a WM\_QUERYWINDOWPARAMS or EM\_GETCHANGED message. Otherwise, fChange is FALSE.

###### Note

This message causes the change state of the edit control to be set to FALSE.

---

##### EM\_QUERYSEL

---

###### Format

```
EM_QUERYSEL
lParam1:      0
lParam2:      0L
Returns:      LOUINT(): INT ichMinSel
              HIUINT(): INT ichMaxSel
```

###### Description

Get the current selection of an edit control. ichMinSel is the byte offset of the first character in the selection, and ichMaxSel is the byte offset of first character after the selection. (The current selection consists of one or more characters if ichMaxSel > ichMinSel. The current selection consists of an insertion point if ichMinSel == ichMaxSel.

---

##### EM\_SET

---



## Format

```
EM_SETSEL
lParam1:      0
LOUINT(lParam2): INT  ichMinSel
HIUINT(lParam2): INT  ichMaxSel
Returns:      OL
```

## Description

Set the current selection to characters within the text with byte offsets less than ichMaxSel and greater than or equal to ichMinSel. (If ichMinSel == ichMaxSel, the current selection becomes an insertion point. If ichMinSel == 0 and ichMaxSel >= the number of characters in the buffer, the entire text is selected.)

---

EM\_SETFONT

---

## Format

```
EM_SETFONT
lParam1:      HANDLE  hFont
lParam2:      OL
Returns:      OL
```

## Description

This message sets the edit control font to hFont (a Font Handle.)

---

EM\_SETTEXTLIMIT

---

## Format

```
EM_SETTEXTLIMIT
LOUINT(lParam1): INT  cchMax;
lParam2:      OL
Returns:      BOOL  fSuccess
```

## Description

This message is used to set the maximum number of characters the user may enter into a text field. EM\_SETTEXTLIMIT returns a boolean fSuccess which is TRUE if the operation was successful, and FALSE if there was not enough memory.

## Notes

This message is intended only to limit the length of lines that result from the user interacting with the edit control. This message also limits the length of text that can result from sending EM\_PASTE, or WM\_SETWINDOWPARAMS messages.

---

## EM\_CUT

---

### Format

EM\_CUT  
lParam1: 0  
lParam2: 0L  
Returns: BOOL fSuccess

### Description

Sends the current selection to the clipboard in CF\_TEXT format, then deletes the selection from the control window. The lParam1 and lParam2 parameters are not used.

---

## EM\_COPY

---

### Format

EM\_COPY  
lParam1: 0  
lParam2: 0L  
Returns: BOOL fSuccess

### Description

Sends the current selection to the clipboard in CF\_TEXT format. The lParam1 and lParam2 parameters are not used.

---

## EM\_CLEAR

---

### Format

EM\_CLEAR  
lParam1: 0  
lParam2: 0L  
Returns: BOOL fSuccess

### Description

This message deletes the current selection.

---

## EM\_PAS

---

### Format

EM\_PASTE  
lParam1: 0  
lParam2: 0L  
Returns: BOOL fSuccess

**Description**

Replaces the current selection with text from the clipboard. (If the the current selection is an insertion point, text is inserted.) Text is inserted only if the clipboard contains data in CF\_TEXT format.

### 4.1.14 Listbox Controls

A listbox control is a window containing a list of items. Each item in a listbox contains a text string (0 or more characters) and a handle. The text string is usually displayed in the listbox window. The handle may be used by the application to refer to other data associated with each item. The control allows the user to make selections from the list and notifies the owner when interesting events occur.

A listbox always contains a scroll bar. If the listbox contains more items than can be displayed in the listbox window rectangle, the scroll bar is enabled. Otherwise, the scroll bar is disabled. (The enabled/disabled state of the scroll bar changes dynamically as items are added to and deleted from the listbox.

The maximum number of items in a listbox control is 32767, but most lists will be much shorter.

The class name to use when creating listbox controls is WC\_LISTBOX.

#### 4.1.14.1 Listbox Control Styles

A listbox can have combinations of the following styles.

---

Style	Meaning
LS_MULTIPLESEL	The listbox control allows the user to more than one item to be selected at any one time. Lists that do not have this style allow only a single selection at any one time.
LS_OWNERDRAW	The listbox has one or more items that will be drawn by the owner. Typically, these items will be represented by bitmaps rather than by text strings.

#### 4.1.14.2 Listbox Control Notification Messages

Listbox controls send the following notification codes to their owner. For a general description of notification codes, see the section on Control Windows.

---

Style	Meaning
-------	---------

LN_SELECT	a new selection has been made in the listbox control. LOUINT(lParam1) contains the index of the selected (or deselected) item.
-----------	--

LN_SETFOCUS	The list has received the input focus.
-------------	--

LN_KILLFOCUS	The listbox control has lost the input focus.
--------------	---

LN_SCROLL	The listbox control is about to scroll.
-----------	---

LN_DBLCLICK	The listbox control been doubleclicked in with the mouse.
-------------	---

---

WM_DRAWITEM	
-------------	--

---

Format	
--------	--

WM_DRAWITEM	
LOUINT(lParam1):	INT idListBox
lParam2:	OI FAR *lpoi
Returns	BOOL fDrawn

idListBox is the window id of the listbox control which is sending the notification. lpoi is a pointer to an owner item, which has the following structure:

```
typedef struct {
    HPS hps;
    RECT rcItem;
    INT iItem;
    HANDLE hItem;
} OI;
```

Description	
-------------	--

This notification is sent to the owner of a listbox each time an item must be drawn. lpoi->iItem contains the item number, and lpoi->hps is the hps which should be used for drawing. lpoi->rcItem contains a rectangle that specifies where to draw the item. The owner must return TRUE if

it actually draws the item, FALSE otherwise. If the item contains text, the owner may return FALSE and the listbox will draw the item itself. (The listbox will draw the item if and only if the item contains text and the owner returns FALSE.)

This message may be used by applications that want to create listboxes containing items that are not represented by text strings. Since listboxes only know how to draw text, this message must be sent to the owner to draw other objects. Typically, `lpoi->hItem` will contain a handle to the object to be drawn.

---

## WM\_MEASUREITEM

---

### Format

```
WM_MEASUREITEM
LOUINT(lParam1):    INT      idListBox
lParam2:            OI FAR *lpoi
Returns:            INT      Height
```

### Description

This notification also is sent to the owner of a listbox containing with `LS_OWNERDRAW` style. When the owner receives this message, it must calculate the height of the item and return that value.

Note - All items in a listbox must have the same height, which must be greater than or equal to the height of the current font.

### 4.1.14.3 Listbox Control States

Listbox controls have only the standard window states. The enabled state of a listbox may be obtained with the `WinIsWindowEnabled()` function.

### 4.1.14.4 Listbox Control Messages

An application can send or post the following messages to a listbox control. (See the general control message section for a description of messages that can be sent to all controls.)

---

## LM\_QUERYITEMCOUNT

---

## Format

```
LM_QUERYITEMCOUNT
lParam1:      0
lParam2:      OL
Returns:      INT cItem
```

## Description

This message returns a count of the number of items in the listbox control.

---

LM\_INSERTITEM

---

## Format

```
LM_INSERTITEM
LOUINT(lParam1):  INT      iPosition
lParam2           LPSTR    lpszItemText
Returns:          INT      iItemActual
```

## Description

This message inserts an item into a list. *iPosition* is the 0-based position in the list where the item should be inserted. If *iPosition* is *LIT\_END* the item is added to the end of the list. If *iPosition* is *LIT\_SORTASCENDING*, it is insertion sorted into the list in ascending order. If *iPosition* is *LIT\_SORTDESCENDING*, the item is inserted in descending order. *lpszItemText* is a pointer to the string to be used as the item text. *iItem* is the actual position where the item was inserted.

## Notes

If the control cannot allocate space to insert the item in the list, it will return *LIT\_MEMERROR*.

---

LM\_SETTOPINDEX

---

## Format

```
LM_SETTOPINDEX
LOUINT(lParam1):  INT      iItem
Returns:          BOOL     fScrolled
```

## Description

This message is used to scroll a particular item to the top of the listbox. *iItem* is the index of the item to be moved to the top. *fScrolled* is *TRUE* if the listbox actually had to scroll to move this item to the top. *fScrolled* is *FALSE* if the item was already at the top, or if *iItem* is not in the list.

---

**LM\_QUERYTOPINDEX**

---

**Format**

```
LM_QUERYTOPINDEX
lparam1:          OL;
lparam2:          OL;
Returns:          INT      iItem;
```

**Description**

This message is get the index value of the item currently at the top of the listbox.

If there are no items in the listbox, LIT\_ NONE is returned.

---

---

**LM\_DELETEITEM**

---

**Format**

```
LM_DELETEITEM
LOUINT(lParam1):  INT iItem
lParam2:          OL
Returns:          INT cItemsLeft
```

**Description**

This message deletes an item from the listbox control. iItem is the (0 based) index of the item to be deleted. cItemsLeft is the number of items remaining in the list after the item is deleted.

---

---

**LM\_SELECTITEM**

---

**Format**

```
LM_SELECTITEM
LOUINT(lParam1):  INT iItem
lParam2:          BOOL fSelect
Returns:          BOOL fSelected;
```

**Description**

This message sets the selection state of a the specified item. If fSelected is TRUE, the item is selected, otherwise the item is deselected.

For single selection listboxes, the previous selection is deselected. If iItem is LIT\_ NONE, no item is selected.

fSelected is TRUE if a selection is actually made, and FALSE otherwise. fSelected will be FALSE if the caller tries to select an item not in the list or

deselects the current selection of a single selection listbox with `iItem == LIT_NONE`.

If an item which is not already selected is deselected, the selected item does NOT get deselected and the message returns `FALSE`.

---

## LM\_GETSELECTION

---

### Format

```
LM_GETSELECTION
LOUINT(lParam1):    INT iItemPrevious;
lParam2:            OL
Returns            UINT iItemSelected
```

### Description

This message is used to enumerate the selected item(s) in a list box. For a single selection listbox, this message always returns the index of the selected item, or `LIT_NONE` if no item is selected.

For multiple selection listboxes, `lParam1` is the previously enumerated selected item. The message returns the index of the next selected item, starting from `lParam1`. If `lParam1` is `LIT_FIRST`, the first selected item is returned.

Returns `LIT_NONE` after all items have been enumerated.

### Example

The following example shows how to count the number of selected items and place their indexes in an array:

```
iItem = LIT_FIRST;
cItems = 0;
while ((iItem = (int)WinSendMsg(hwndListBox,
    LM_GETSELECTION, iItem,
    OL)) != LIT_NONE) {
    rgItems[cItems++] = iItem;
}
```

---

## LM\_SETITEMTEXT

---

### Format

```
LM_SETITEMTEXT
LOUINT(lParam1):    UINT iItem;
lParam2:            LPSTR lpszBuffer;
Returns:            BOOL fSuccess;
```



**Description**

Sets the text of the specified item by copying the text from `lpszBuffer` to the listbox. Returns `TRUE` if successful, `FALSE` otherwise.

---

**WM\_SETITEMHEIGHT**

---

**Format**

```
WM_SETITEMHEIGHT
LOUINT(1Param1):    INT      NewHeight;
1Param2:            OL;
Returns:            OL;
```

**Description**

This message is used to change the height of the items in a listbox. It is easier to send this message than to destroy and re-create the listbox with items at the new height.

---

**LM\_QUERYITEMTEXTLENGTH**

---

**Format**

```
LM_QUERYITEMTEXTLENGTH
LOUINT(1Param1):    UINT  iItem
1Param2:            OL
Returns:            INT   cbItem
```

**Description**

Gets the size in bytes of the text for the specified item. Returns 0 if `iItem` does not exist, or some other error occurs.

---

**LM\_QUERYITEMTEXT**

---

**Format**

```
LM_QUERYITEMTEXT
LOUINT(1Param1):    UINT  iItem;
HIUINT(1Param1):    INT   cbBuffer;
1Param2:            LPSTR lpszBuffer;
Returns:            INT   cchString;
```

**Description**

Copies the specified item text from the listbox to buffer pointed to by `lpszBuffer`. `cbBuffer` specifies the maximum number of bytes that may be copied to the buffer. If `cbBuffer` is 0, the entire text string is copied.

The message returns the length of the string copied, not including the null termination character. The length uint is overwritten by the string.

The size of the item can be determined with the LM\_QUERYITEMTEXTLENGTH message.

---

## LM\_SETITEMHANDLE

---

### Format

```
LM_SETITEMHANDLE
LOUINT(1Param1):    UINT    iItem;
1Param2:             HANDLE  hItem;
```

### Description

Sets the item handle for the specified item to hItem.

---

## LM\_QUERYITEMHANDLE

---

### Format

```
LM_QUERYITEMHANDLE
LOUINT(1Param1):    UINT    iItem;
Returns:            INT     hItem;
```

### Description

Returns hItem, the handle associated with the specified item.

---

## LM\_SEARCHSTRING

---

### Format

```
LM_SEARCHSTRING
LOUINT(1Param1):    UINT    cmd
HIUINT(1Param1):    UINT    iItemStart;
1Param2:            LPSTR   lpszSearch;
Returns:            UINT    iItemFound;
```

### Description

This message returns the item index of the next item whose string matches the string in \*lpszSearch. All items are enumerated: searching wraps around at the end, starting over again at the first item. Searching starts at the first item AFTER iItemStart; thus the value from a previous LM\_SEARCHSTRING message may be used as the starting point in order to find the next matching item. If iItemStart is LIT\_FIRST, searching starts

from the first item in the listbox.

cmd contains one of the following values that specifies how a match is to be found. (These values can be combined using the OR operator.)

---

**LSS\_SUBSTRING**

The item is matched if it contains a substring that matches \*lpszSearch.

**LSS\_PREFIX**

The item is matched if \*lpszSearch matches the initial characters of the item.

**LSS\_CASESENSITIVE**

Corresponding characters must be the same case to match.

Returns LIT\_NONE if no match was found.

Notes To determine whether a matched item is unique, send another LM\_FIRSTSTRING message using the matched item as a starting point. If a different item is returned, the item is not unique.

### 4.1.15 Scroll Bar Controls

Scroll bars are controls that are used to indicate that additional information can be displayed in a window, logically to the left or right for horizontal scroll bars, logically above or below for vertical scroll bars. The user interface for scroll bars allows for scrolling one 'unit' or one 'page' at a time, or alternately picking up the scroll bar 'thumb' and sliding it to a position in the scroll bar that indicates a logical position in the program.

If the user holds the mouse or key button down that caused the scroll bar action, the messages will auto-repeat at a rate determined by the value of the SV\_SBREPEATTIME system value.

The default position for a vertical scroll bar is aligned to the right edge of the frame window, between the application menu and the size box. A horizontal scroll bar is, by default, aligned to the bottom edge of the frame window, between the left edge and the size box.

A standard scroll bar has a different appearance when disabled, as determined by user interface.

The class value used to identify the scroll bar class is WC\_SCROLLBAR.

The standard IDs used to identify the vertical and horizontal scroll bars in frame windows are:

---

ID	Description
FID_VERTSCROLL	Vertical scroll control ID
FID_HORZSCROLL	Horizontal scroll control ID

Scroll bars are created with a default resolution of 0 to 100.

#### 4.1.15.0.1 Scroll Bar Styles

---

Style	Description
SBS_HORZ	Create a horizontal scroll bar
SBS_VERT	Create a vertical scroll bar

#### 4.1.15.1 Scroll Bar Notification Messages

---

##### WM\_VSCROLL

---

###### Format

```
WM_VSCROLL
lParam1:        ULONG idCtl;
LOUINT(lParam2): int pos;
HIUINT(lParam2): UINT cmd;
Returns:        OL
```

---

##### WM\_HSCROLL

---

###### Format

```
WM_HSCROLL
lParam1:        ULONG idCtl;
LOUINT(lParam2): int pos;
HIUINT(lParam2): UINT cmd;
Returns:        OL;
```

###### Description

These two messages are posted to the scroll bar owner's queue by the window manager. idCtl

contains the control window id. cmd contains a scroll command, which can be one of the following:

#### Scroll Commands

Command

##### SB\_LINEUP

This is sent if the user hits the up or left arrow of the scroll bar with the mouse, or if the user hits the VK\_UP key.

##### SB\_LINELEFT

Save value as SB\_LINEUP, included for completeness.

##### SB\_LINEDOWN

This is sent if the user hits the down or right arrow of the scroll bar with the mouse, or if the user hits the VK\_DOWN key.

##### SB\_LINERIGHT

Same value as SB\_LINEDOWN, included for completeness.

##### SB\_PAGEUP

This is sent if the user hits the halftone area above or to the left of the scroll thumb, or if the user hits the VK\_PAGEUP key.

##### SB\_PAGELLEFT

Same value as SB\_PAGEUP, included for completeness.

##### SB\_PAGEDOWN

This is sent if the user hits the halftone area below or to the right of the scroll thumb, or if the user hits the VK\_PAGEDOWN key.

##### SB\_PAGERIGHT

Same value as SB\_PAGEDOWN, included for completeness.

##### SB\_THUMBTRACK

If the user grabs hold of the scroll bar thumb with the mouse, this is sent every-time the thumb position changes.

##### SB\_THUMBPOSITION

If the user grabbed hold of the scroll bar thumb with the mouse, this message is sent once the user lets go.

**SB\_ENDSCROLL**

Sent when the user has finished scrolling, only if the user wasn't doing an absolute thumb positioning. LOUINT(IParam2) contains TRUE if the cursor was inside the tracking rectangle when the mouse button was released, FALSE otherwise.

LOUINT(IParam2) holds the scroll position only if the user was moving the thumb with the mouse (SB\_THUMBTRACK and SB\_THUMBPOSITION).

*4.1.15.1.1 Scroll bar control messages*

The following messages may be sent by the application to control or query the scroll bar. Scroll bars also respond to the WM\_SET/QUERYWINDOWPARAM messages.

---

**SBM\_SETSCROLLBAR**

---

**Format**

SBM\_SETSCROLLBAR  
LOUINT(lParam1): int pos;  
LOUINT(lParam2): int posFirst;  
HIUINT(lParam2): int posLast;  
Returns: OL

**Description**

This message is used to set the scroll bar range and position information in the scroll bar control.

LOUINT(lParam2) and HIUINT(lParam2) delimit the range information, their values being inclusive. lParam1 is the position of the thumb within this range. If this position is illegal, the thumb will be moved to the nearest position within the range. The scroll bar is redrawn to reflect these changes.

The application usually sends this message when either it's initializing a scroll bar or it's client window is changing in size. These are cases when both the range and position information need to be sent at once.

---

**SBM\_SE**

## Format

```
SBM_SETPOS
LOUINT(lParam1): int pos;
lParam2:         OL
Returns:         OL
```

## Description

This message sets the thumb position of the scroll bar control. lParam1 is the position of the thumb within this range. If this position is illegal, the thumb will be moved to the nearest position within the range. The scroll bar is redrawn to reflect these changes.

---

SBM\_QU

## Format

```
SBM_QUERYPOS
lParam1:         0
lParam2:         OL
Returns:         int pos;
```

## Description

This message simply returns the thumb position for this scroll bar.

---

SBM\_QUERYRANGE

## Format

```
SBM_QUERYRANGE
lParam1:         0
lParam2:         OL
LOUINT(return):  int posFirst;
HIUINT(return):  int posLast;
```

## Description

This message is used to get the scroll range information from the scroll bar control. LOUINT(return) and HIUINT(return) delimit the scroll bar range.

#### 4.1.16 Menu Controls

A menu control is a small child or popup window that contains a list of items. These items can be represented by text strings, separators, bitmaps or menu buttons. Menu templates can be loaded as resources and the menu can be created automatically when the parent window is created. Also, the application can create a menu by calling `WinCreateWindow` with a class `WC_MENU`. The application can build the menu dynamically by sending `MM_INSERTITEM` messages, etc. An application can alter menus by sending messages to it.

Menus allow you to select one of the items in the list using the mouse or the keyboard interface. When you make this selection, the menu notifies its parent by posting (PostMsg) a WM\_COMMAND or WM\_SYSCOMMAND message and a unique ID representing the user's selection.

Menus automatically resize themselves as items are added and removed.  
Menus are automatically destroyed when their owner is destroyed.

Typically, an application has an action bar menu and several popup sub-menus. The action bar is normally visible, and is a child window in the parent window frame. The popups are normally hidden and become visible when selections are made on the action bar.

#### 4.1.16.1 Action Bar Layout

The items displayed in an action bar are displayed as follows:

1. Choices are left-aligned in the action bar. That is, they begin near the left side of the window and are adjacent to each other.
2. Each choice is displayed with a leading and trailing blank.
3. Choices are listed horizontally but may be reformatted to multiple rows as the size of the window in which it is shown is decreased.
4. The action bar is separated from the panel body by a solid line.

#### 4.1.16.1.1 Action Bar Appearance Examples

The following example shows typical content and layout for the action bar in an application window:

Small Editor						--Window Title
Properties	Block	Search	Format	Exit	%F1=Help	--Action Bar
GHI Memo						full width of window



```

January 24, 1986

MEMO TO:      A.B. Curtis
FROM:        D.E. Fitzgerald
SUBJECT:     Growing Households Inventory

Abe,

    We have just completed the latest households invento
and it really looks great.  It will allow us to maintain

```

**Figure 4.3 Action Bar**

The next example shows how the action bar is reformatted when the window size is decreased.

```

+-----+
|      Small Editor      |
+-----+-----+
|> Properties           %|<
|  Block Search      #  |<
|  Format Exit       |<
+-----+-----+
|  GHI Memo  More:      v  |
|
|  January 24, 1986
|
|  MEMO TO:      A.B. Curtis
|    FROM:      D.E. Fitzge
|  SUBJECT:     Growing Hou
|
|  Abe,
|
|    We have just complete
|  and it really looks great
+-----+

```

**Figure 4.4 Reformatted Action Bar**

#### 4.1.16.2 Menu Control Styles

A menu control can have combinations of the following styles:

---

Menu Control Styles

MS\_ACTIONBAR

The items in the list are displayed side by side. This style is used to implement a Top Level Menu. Menus that do not have this style are displayed in one or more columns and are

submenus associated with an action bar.

All menu controls have styles `WS_SYNCPAINT` and `WS_PARENTCLIP`.

#### 4.1.16.3 Menu Items

There are two types of menu items: Submenu and Command items. A sub-menu item contains a reference to a sub-menu which will be displayed when the user selects that item. A command item contains a value which is returned by the menu when the item is selected.

---

Field	Meaning
-------	---------

Handle/ID	
-----------	--

	If the item is submenu, this field contains the window handle of a submenu which is displayed when the item is selected. If the item is command, this field contains a unique ordinal value identifies the item.
--	--

Display Object	
----------------	--

	This object is displayed to represent the item. It can be a string, bitmap, separator or menu button.
--	---

#### 4.1.16.4 Menu Item Styles

A menu item can have combinations of the following styles.

---

Menu Item Styles
Style

<code>MIS_SUBMENU</code>
--------------------------

The item is submenu. When the user selects this type of item, a submenu is displayed from which the user must make further selection. Items that are not submenu items are Command Items.
---

<code>MIS_SEPARATOR</code>
----------------------------

The display object is a horizontal dividing line. This type of item can only be used in popup menus. This type of item cannot be enabled, checked, disabled, highlighted or selected by the user. The functional object is <code>NULL</code> when this style is specified.
--

<code>MIS_BITMAP</code>
-------------------------

The display object is a bitmap.
---------------------------------

**MIS\_ TEXT**

The display object is a text string.

**MIS\_ BUTTONSEPARATOR**

The item is a menu button. Any menu may have zero, one or two items of this type. These are the last items in a menu and are automatically displayed after a separator bar. The user can not cursor to these items, but he can select them with the mouse or select them with the appropriate key.

**MIS\_ BREAK**

The item is the last one in a row or column. The next item is displayed at the beginning of a new row or column.

**MIS\_ BREAKSEPARATOR**

Same as MIS\_ BREAK, except that it draws a separator between rows or columns.

**MIS\_ NODISMISS**

If this item is selected, the popup menu containing this item should not be hidden before notifying the application window.

**MIS\_ SYSCOMMAND**

If this item is selected, the menu notifies the owner by posting a WM\_SYSCOMMAND message rather than a WM\_COMMAND message.

**MIS\_ OWNERDRAW**

Items with this style are drawn by the owner. WM\_DRAWITEM and WM\_MEASUREITEM notification messages are sent to the owner to draw the item or determine its size.

**MIS\_ HELP**

If the item is selected, the menu notifies the owner by posting a WM\_HELP message rather than a WM\_COMMAND.

**MIS\_ STATIC**

This type of item exists for information purposes only. It cannot be selected with the mouse or keyboard.

#### *4.1.16.4.1 Placing Help selection in Action Bar*

It is recommended that a Help item is put into the Action Bar to provide the Mouse user with a means of getting Help on the Action Bar or its related client window. The item should read "F1=Help" and always appear at the top right of the action bar.

The method by which the placement and function of the F1=Help item is achieved in the Menu Bar is as follows:

- The item should be the last item in the template for the Menu Bar itself.
- It should not have an associated Pull-Down menu associated with it.
- It should be MIS\_ TEXT|MIS\_ BUTTONSEPARATOR style and the string should be *F1=Help*.
- The MIS\_ BUTTONSEPARATOR style will put the item in the top right of the menu bar, separated from the other items by a separator bar. It can only be used by the Mouse and cannot be cursoried to from the Keyboard. Note that the F1 key will achieve the same function as the menu item for the keyboard button.

#### 4.1.16.5 Menu Item Attributes

Menu Items have the following attributes. Applications can get and set the state of these attributes by sending MM\_ GETITEMSTATE and MM\_ SETITEMSTATE messages.

---

Menu Item Attributes	Attribute
----------------------	-----------

MIA_ HILITED
--------------

The state of this attribute is TRUE if and only if the item is selected.
--

MIA_ ENABLED
--------------

If the state of this attribute is TRUE, the item is enabled and can be selected by the user. If the state is FALSE, it is disabled and cannot be selected. The item is halftoned.
---

MIA_ CHECKED
--------------

A checkmark appears next to the item if the state of this attribute is TRUE.
--

#### 4.1.16.6 Mnemonics and Mnemonic Highlighting.

A *Mnemonic* is a single letter which can be typed on the keyboard as a fast way of selecting a Text item within a menu. A mnemonic comes into operation when the Input Focus is in a Menu window - either the Action Bar or one of the Pull-Down menus. The scope of the mnemonic is the Input Focus window - so that an action bar and a pull-down can use the same mnemonic letter for different actual items.

Where mnemonics are used, it is strongly suggested that **ALL** the items within one menu window are given a mnemonic letter. The letter chosen should be unique for each item within that menu and must occur within the text string representing the item. So, for example, if a menu has the

items 'Edit', 'File' and 'Erase', the mnemonic letters chosen might be 'E', 'F' and 'R' respectively.

Mnemonic letters within menu items are highlighted by means of an underscore, which guides the user's eye. However, after a little use of a menu with Mnemonics, the user should be able to select sequences of items rapidly directly from the keyboard without even looking at the screen.

The application defines the mnemonic letter for each item by placing a 'tilde' character into the text string, immediately in front of the mnemonic character. When the menu is displayed, the tilde characters are not displayed and the mnemonic characters are highlit. If it is necessary to display a tilde character in a menu item, then the application should put 'tilde tilde' into the initial text string. This applies both to items defined via resource files and to those defined directly from the application.

Note that if more than one mnemonic character is defined for a single item, the first mnemonic character in the string is accepted.

Where the mnemonic character is a letter, both lower and upper case characters are accepted as mnemonics from the keyboard.

#### **4.1.16.7 Menu Data Structures**

##### *4.1.16.7.1 Item Data Structure*

When an application queries or sets the item in a menu, it is transfered in the following data structure:

```
typedef struct {  
    INT      iPosition;  
    UINT     rgfStyle  
    UINT     id;  
    HWND     hwndSubMenu;  
    HANDLE   hItem;  
} MENUITEM;
```

The iPosition field contains the position index. The rgfStyle field contains flags that describe the behavior and appearance of the item. The id field contains a unique value that identifies the item. hwndSubMenu contains the window handle of the submenu associated with the particular item if there is one. Otherwise, hwndSubMenu contains 0. For all objects except MIS\_TEXT, hItem contains a handle to the display object. For Text objects, hItem is NULL. The application must send a separate message to query or set the item text.

#### 4.1.16.7.2 Menu Template Data Structure

Menu templates are data structures that define menus that will be created. Menu templates can be loaded as resources or created dynamically. Templates loaded as resources can not contain references to bitmaps or owner-draw items, shown below. A menu template consists of a sequence of variable length records. Each record in a template defines an item. If an item contains a reference to a submenu, the template that defines that submenu is nested after the definition of that particular item.

```
typedef struct tagMT
{
    UINT cmti;
    UINT codepageid;
    MTI  rgmti(cmti);
    MT;
```

---

cmti        contains the count of menu template items.

codepageid  
          is the identifier of the codepage used for the text items within the menu (but not any submenus, which each have their own codepages). Note that this MUST be 850 for the present.

rgmti      is a variable sized array of menu template items (described below.)

```
typedef struct tagMTI
{
    UINT rgfStyle;
    UINT idItem;
    if (rgfStyle AND MIS_BITMAP)
        CHAR szItemString{?};
    if (rgfStyle AND MIS_OWNERITEM)
        VOID;
    if (rgfStyle AND MIS_TEXT)
        CHAR szItemString{?};
    if (rgfStyle AND MIS_SEPARATOR)
        VOID;
    if (rgfStyle AND MIS_SUBMENU)
        MT MenuTemplate;
    MTI;
```

---

rgfStyle   contains a combination of menu item styles (MIS\_\*) combined with the OR operator.

idItem     contains a unique non-negative integer value which identifies the item.

Following idItem is a variable data structure whose format depends upon the value of rgfStyle:

- If `rgfStyle` contains `MIS_ TEXT`, it is a 0-terminated string.
- If `rgfStyle` contains `MIS_ BITMAP`, it is a 0-terminated string.

For `MIS_ BITMAP` menu items, the item text string may be used to derive the resource ID from which a bitmap will be loaded. There are 3 cases:

1. The first byte is `NULL` - ie no resource is defined and it is assumed that the application will subsequently provide a bitmap handle for the item.
2. First byte is `0xff`, 2nd byte is the low byte of the resource ID, 3rd byte is the hi byte of the resource ID.
3. First character is `"#"`, subsequent characters make up the decimal text representation of the resource ID.

The resource is assumed to reside in the resource file of the current process.

If the string is empty or does not follow the format above, no resource is loaded.

- If `rgfStyle` contains `MIS_ OWNERDRAW`, it is expected that the application will subsequently provide a handle to some application-defined object for the item.
- If `rgfStyle` contains `MIS_ SEPARATOR`, it is `VOID`.
- If `rgfStyle` contains `MIS_ SUBMENU`, it contains a complete submenu data structure (ie a menu template)

#### 4.1.16.8 Menu Notification Messages

Menu controls send the following notification codes to their owner. For a general description of notification codes, see the section on Control Windows.

---

Menu Notification Codes  
Code

`WM_ INITMENU`

---

Format

```
WM_INITMENU
LOUINT(lParam1):    INT        idMenu
lParam2:            HANDLE     hwndMenu
```

This notification is sent to the owner whenever a menu is about to become "active." (The menu window is not really activated, and it does not receive the focus, it just looks and acts like it is.) This

notification is often used to let application change the state of menu items before the menu is displayed. LOUINT(lParam1) contains the id of the menu being initialized and lParam2 contains the menu window handle.

## WM\_MENUSELECT

---

### Format

```
WM_MENUSELECT
LOUINT(lParam1):    UINT        idItem;
HIUINT(lParam1):    BOOL        fPostCommand
lParam2:            HWND        hwndMenu
```

An item in the menu was selected. In this case, the menu posts a WM\_COMMAND or a WM\_SYSCOMMAND message to its owner. LOUINT(lParam1) contains the ID of the selected item. HIUINT(lParam1) contains BOOL fPostCommand. lParam2 contains the menu window handle.

If fPostCommand is TRUE, then the item being selected will cause a WM\_COMMAND or WM\_SYSCOMMAND message to be posted, and the menu possibly to be dismissed. This is essentially a synchronous command notification. If the message returns TRUE, then the command will be posted as usual, and the menu will be dismissed (unless the currently selected item has attribute MIS\_NODISMISS.) If FALSE is returned, then no message is posted, and the menu is not dismissed.

## WM\_DRAWITEM

---

### Format

```
WM_DRAWITEM
LOUINT(lParam1):    INT        idMenu
lParam2:            OWNERITEM FAR *lpoi
```

---

idMenu is the window id of the menu control which is sending the notification.

lpoi is a pointer to an owner item, which has the following structure:

```
typedef struct tagOWNERITEM
{
    HPS hps;
    UINT rgfStyle;
    RECT rcItem;
    UINT idItem;
    HANDLE hItem;
}
```



```
} OWNERITEM;
```

This notification is sent to the owner of a menu anytime it is necessary to draw an item that has item style `MIS_OWNERDRAW`. When the menu owner receives this message it is responsible for drawing the specified item using `hps = lpoi->hps` inside the rectangle `lpoi->rcItem`. If the item has style `MIS_TEXT`, `lpoi->hItem` is `NULL`, and the owner can get the text for the item by sending a `MM_QUERYITEMTEXT` message. Otherwise, `hItem` is a handle to a display object that will be interpreted by the owner.

`rgfStyle` is the same as the style field in the `MENUITEM` data structure.

---

### WM\_MEASUREITEM

Format

```
WM_MEASUREITEM
LOUINT(lParam1):      INT      idMenu
lParam2:              OWNERITEM FAR *lpoi
```

This notification also is sent to the owner of a menu containing items with `MIS_OWNERDRAW` style. When the owner receives this message, it must calculate the size of the item and store in `lpoi->rcItem.xRight` and `lpoi->rcItem.yTop`. The `lpoi->rcItem.xLeft` and `lpoi->rcItem.yBottom` should not be set by the owner.

#### 4.1.16.9 Menu Control Messages

An application can send or post the following messages to a menu control.

See the general control message section for a description of messages that can be sent to all controls.

---

### MM\_STARTMENU MODE

Format

```
MM_STARTMENU MODE
LOUINT(lParam1):      BOOL      fMouse
lParam2:              POINT     ptMouse
```

**Description**

This message is used to begin menu selection. It is sent to the menu when the user presses the menu key. `fMouse` is `TRUE` if a mouse event has caused the message. If `fMouse` is `TRUE`, `ptMouse` contains the initial mouse coordinates.

---

**MM\_ENDMENU****Format**

`MM_ENDMENU`

**Description**

This message is used to end menu selection. When the menu receives this it hides the menu window if it is a popup window. `WM_CANCELMODE` has the same effect as `MM_ENDMENU`.

---

**MM\_INSERTITEM****Format**

`MM_INSERTITEM`  
`lParam1: MENUITEM FAR *lpItem`  
`lParam2: LPSTR lpszText`  
`Returns: int iItemActual`

**Description**

This message inserts an item into a menu.

`lpItem` points to an `MENUITEM` data structure containing the item to be inserted. (This data structure includes an `iPosition` field. If `iPosition` is `MIT_END`, the item is added to the end of the list.

`lpszText` points to a text string if the item style includes `MIS_TEXT`.

`iItemActual` is the actual position where the item was inserted.

**Notes** If the control cannot allocate space to insert the item in the list, it will return `MIT_MEMERROR`.

---

**MM\_DELETEITEM**

---

**Format**

```
MM_DELETEITEM
LOUINT(1Param1)  UINT      idItem
HIUINT(1Param1)  BOOL      fIncludeSubmenus
Returns:         INT      cItemsLeft
```

**Description**

This message deletes an item from the menu control. `idItem` is the id of the item to be deleted. `cItemsLeft` is the number of items remaining in the list after the item is deleted. Any display object or submenu referenced by the item is destroyed.

If `fIncludeSubmenus` is `TRUE`, and if the window which receives this message does not have an item with the specified id (`idItem`), the submenus of this menu will be searched for an item with a matching id. If a match is found, the operation will be performed on the submenu.

---

**MM\_REMOVEITEM**

---

**Format**

```
MM_REMOVEITEM
LOUINT(1Param1)  UINT      idItem
HIUINT(1Param1)  BOOL      fIncludeSubmenus
Returns:         INT      cItemsLeft
```

**Description**

Same as `MM_DELETEITEM`, except does not destroy the display object or submenu.

---

**MM\_SELECTITEM**

---

**Format**

```
MM_SELECTITEM
LOUINT(1Param1):  UINT  idItem
HIUINT(1Param1):  BOOL  fIncludeSubmenus
LOUINT(1Param2):  BOOL  fSelected
HIUINT(1Param2):  BOOL  fDismiss
Returns:          BOOL  fSuccess
```

**Description**

This message sets the selection state of the specified item. if `fDismiss` is `TRUE`, then `WM_SYSCOMMAND` or `WM_COMMAND` is

posted, and the menu is dismissed, if the item is not `MIS_NODISMISS`. If `fSelected` is `TRUE`, the item is selected. If `fSelected` is `FALSE`, the item is deselected. `fSuccess` is `TRUE` if a selection is actually made, and `FALSE` otherwise.

The application can set the selection to `NULL` by setting `idItem = MIT_NONE`, in which case `fSuccess` will be `TRUE`.

If `fIncludeSubmenus` is `TRUE`, and if the window which receives this message does not have an item with the specified id (`idItem`), the submenus of this menu will be searched for an item with a matching id. If a match is found, the operation will be performed on the submenu.

---

## MM\_GETSELITEMID

---

### Format

<code>MM_GETSELITEMID</code>		
<code>lParam1:</code>	<code>OL;</code>	
<code>lParam2:</code>	<code>OL;</code>	
Returns	<code>UINT</code>	<code>idItemSelected</code>

### Description

This message returns the ID of the currently selected item (even if in a submenu.)

---

## MM\_QUERYITEM

---

### Format

<code>MM_QUERYITEM</code>		
<code>LOUINT(lParam1):</code>	<code>UINT</code>	<code>idItem</code>
<code>HILUINT(lParam1):</code>	<code>BOOL</code>	<code>fIncludeSubmenus</code>
<code>lParam2:</code>	<code>MENUITEM</code>	<code>*lpItem</code>

### Description

Copies the specified item from the menu to the buffer pointed to by `lpItem`.

If `fIncludeSubmenus` is `TRUE`, and if the window which receives this message does not have an item with the specified id (`idItem`), the submenus of this menu will be searched for an item with a matching id. If a match is found, the operation will be performed on the submenu.

Note This message does not retrieve the text for items with style `MIS_ TEXT`. The application must use `MM_ QUERYITEMTEXT`.

---

## `MM_ QUERYITEMTEXT`

---

### Format

```
MM_QUERYITEMTEXT
LOUINT(1Param1):    UINT    idItem
HIUINT(1Param1):    INT     cchBufferMax
1Param2:            LPSTR   lpszBuffer
```

### Description

This message gets the text string which is used for a menu item's display object, if it has style `MIS_ TEXT`.

---

`idItem` is the item id

`cchBufferMax`  
specifies the size of the buffer.

`lpszBuffer`  
points to the buffer to receive the zero terminated string.

---

---

## `MM_ QUERYITEMTEXTLENGTH`

---

### Format

```
MM_QUERYITEMTEXTLENGTH
LOUINT(1Param1):    UINT    idItem
Returns            INT     cchItem
```

### Description

Returns the length of the text string for `MIS_ TEXT` items, not including the NULL terminator. Returns 0 if not `MIS_ TEXT`.

---

---

## `MM_ SETITEMHANDLE`

---

### Format

```
MM_SETITEMHANDLE
LOUINT(1Param1):    UINT    idItem
1Param2:            HANDLE  hItemNew
```

### Description

This message is used to change the bitmap or handle of a non-`MIS_ TEXT` item. `idItem` is the item

Id, and hItemNew contains the handle to the new display object.

---

## MM\_SETITEMTEXT

---

### Format

```
MM_SETITEMTEXT
LOUINT(lParam1):   UINT      idItem
lParam2:           LPSTR    lpszText
```

### Description

This message is used to change the text string display object for items with style MIS\_TEXT.

---

lpszText

points to a zero terminated text string to be used as the new display object.

---

---

## MM\_SETITEM

---

### Format

```
MM_SETITEM
HIUINT(lParam1):  BOOL      fIncludeSubmenus
lParam2:          MENUITEM  *lpItem
```

### Description

This message copies the contents of \*lpItem to the menu item with the same id.

If fIncludeSubmenus is TRUE, and if the window which receives this message does not have an item with the specified id (idItem), the submenus of this menu will be searched for an item with a matching id. If a match is found, the operation will be performed on the submenu.

---

---

## MM\_ITEMPOSITIONFROMID

---

### Format

```
MM_ITEMPOSITIONFROMID
LOUINT(lParam1):   UINT      idItem
HIUINT(lParam1):   BOOL      fIncludeSubmenus
Returns:           INT       iPosition
```

### Description

Given the ID of an Item, this message returns the (0 based) position of the item in the menu. If there

is no item in the menu with this id, the message returns MIT\_NONE.

If fIncludeSubmenus is TRUE, and if the window which receives this message does not have an item with the specified id (idItem), the submenus of this menu will be searched for an item with a matching id. If a match is found, the operation will be performed on the submenu.

---

### MM\_ITEMIDFROMPOSITION

---

#### Format

```
MM_ITEMIDFROMPOSITION
LOUINT(lParam1):    INT   iPosition
Returns:           UINT  idItem
```

#### Description

Given the position, this message returns the ID of the item at that position. If the position is not a valid position, this message returns MIT\_NONE.

---

### MM\_QUERYITEMCOUNT

---

#### Format

```
MM_QUERYITEMCOUNT
Returns: int cItem
```

#### Description

This message returns a count of the number of items in the menu control.

---

### MM\_QUERYITEMATTR

---

#### Format

```
MM_QUERYITEMATTR
LOUINT(lParam1):    UINT  idItem
HIUINT(lParam1):    BOOL   fIncludeSubmenus
LOUINT(lParam2)     UINT  rgfAttributeMask
Returns:            UINT  rgfState
```

#### Description

Get the current state of the attributes of menu item. idItem is the ID of the item. rgfAttributeMask is a bit mask indicating the attribute values to get. Returns the current state values that correspond to the state mask.

If `fIncludeSubmenus` is `TRUE`, and if the window which receives this message does not have an item with the specified id (`idItem`), the submenus of this menu will be searched for an item with a matching id. If a match is found, the operation will be performed on the submenu.

---

## MM\_SETITEMATTR

---

Format

```
MM_SETITEMATTR
LOUINT(lParam1): UINT      idItem
HIUINT(lParam1): BOOL      fIncludeSub'
```

### 4.1.17 Caret Manager

This section describes the functions used to create, display, move, and destroy the system caret. The system caret is a rectangle that can be moved to any location on the display screen. The caret is typically used in text applications to mark the location where characters will be inserted.

The size and position of a caret are specified in window coordinates, relative to the window that the caret is in.

The caret is clipped in a window as with all other drawing to that window. The caret is automatically hidden and shown by `WinScrollWindow()`, hidden by `WinBeginPaint()`, and shown by `WinEndPaint()`.

#### 4.1.17.1 Caret Data Structures

```
typedef struct {
    HWND hwnd;
    INT x, y;
    INT cx, cy;
    UINT rgf;
    INT iHideLevel;
} CARETINFO;
```

#### 4.1.17.2 Caret Routines

---

##### WinCreateCaret

---

Format

```
BOOL WinCreateCaret(hwnd, x, y, cx, cy, rgf)
HWND hwnd;
```



```
INT    x;  
INT    y;  
INT    cx;  
INT    cy;  
UINT   rgf;
```

#### Description

This function creates a caret for window defined by `hwnd`. The caret has its position defined by the coordinates `x` and `y`, its size by `cx` and `cy`. These coordinates are in window coordinates, relative to `hwnd`.

The `rgf` parameter specifies the appearance of the caret, and whether the `cx` and `cy` parameters are interpreted.

The appearance of the caret is specified by `rgf`. This flag can be any combination of the first 5 of the following flags. The last flag bit is used to set the position of the caret.

Returns `TRUE` if successful, `FALSE` otherwise.

---

Caret flag bits

Value

`CARET_SOLID`

The caret is solid.

`CARET_HALFTONE`

Create a halftone caret.

`CARET_RECT`

Solid rectangular caret.

`CARET_FRAME`

Rectangular frame.

`CARET_FLASH`

The caret flashes.

`CARET_SETPOS`

Set a new caret position. `cx` and `cy` are ignored. Used when a caret has already been created.

If `cx` or `cy` is 0, then the system nominal border width or height is used. These values can be obtained by calling `WinGetSysValue` with `SV_CXBORDER` and `SV_CYBORDER` as indexes. The width of a text caret is usually set to 0. This is preferable to giving 1 as the `cx` argument, since on a high resolution device a width of 1 may produce a very thin line.

If the flag `CARET_SETPOS` is specified the caret must already exist and size and appearance flags are

ignored.

The caret is initially hidden. The `WinShowCaret` function must be called to display the caret.

Only one caret is available at a time, so it is not possible to have two carets flashing. An application should create a caret only when it has the keyboard focus or is the active window. Creating carets at other times may stop the caret from flashing in another application. Similarly, when an application becomes inactive or loses the focus, its caret should be destroyed.

---

## `WinDestroyCaret`

---

### Format

```
BOOL WinDestroyCaret (hwnd)
HWND hwnd;
```

### Description

This function destroys the current caret, if it belongs to the specified window. Has no effect if the caret does not belong to `hwnd`.

Returns `TRUE` if successful, `FALSE` otherwise.

---

## `WinShowCaret`

---

### Format

```
INT WinShowCaret (hab, hwnd, fShow)
HWND hwnd;
BOOL fShow;
HAB hab;
```

### Description

This function displays (`fShow` is `TRUE`) or removes (`fShow` is `FALSE`) a caret created with the `WinCreateCaret` function in the same window specified by `hwnd`.

If the caret has been removed several times without any intervening calls to display it, an equal number of `WinShowCaret` calls with `fShow` being `TRUE` must be made before the caret is redisplayed.

This function returns the caret show level, after the caret is shown or hidden. Returns `iHideLevel`, which is 0 if the caret is visible, 1 or greater if the caret is hidden.

---

**WinQueryCaretInfo**

---

**Format**

```
BOOL WinQueryCaretInfo(hab, lpCaretInfo)
HAB hab;
CARETINFO FAR *lpCaretInfo;
```

**Description**

Information about the current caret is returned in \*lpCaretInfo. The fields of this structure correspond to the parameters to WinCreateCaret(), except that the rgf field never includes the CARET\_SETPOS bit.

The size and position of the caret are returned in window coordinates relative to lpCaretInfo->hwnd.

Returns TRUE if a caret exists, FALSE otherwise. If no caret exists, \*lpCaretInfo is not changed.

iHideLevel is a variable that is incremented by WinShowCaret(hwnd,FALSE) and decremented by WinShowCaret(hwnd,TRUE). The caret is visible only if iHideLevel is 0.

### **4.1.18 Mouse Cursor**

This section describes the functions that affect the mouse cursor pointing device.

The appearance of the cursor is defined by a bitmap. The bitmap must be a certain size, defined by the system. There are two possible sizes: the system cursor size, and the system icon size. These values may be obtained with GetSysValue().

The bitmap is always twice as tall as either the system cursor or system icon size. The bitmap is divided into two parts: the XOR mask and the AND mask. The cursor or icon is drawn by first ANDing the screen with the top half of the bitmap, and then XORing with the bottom half of the bitmap.

#### **4.1.18.1 Mouse Cursor Data Structures**

A cursor is identified by a "cursor handle":

```
typedef HANDLE HCURSOR;
```

The CURSORINFO data structure describes a cursor:

```
typedef struct {  
    INT xHotspot;  
    INT yHotspot;  
    HBITMAP hbmCursor;  
} CURSORINFO;
```

The hbmCursor field is a bitmap handle to be used for the cursor image.  
The

The xHotspot and yHotspot fields are the position within the the cursor of the "hot spot". This might be the tip of an arrow cursor, or the center of a crosshair cursor.

---

## WinLoadCursor

---

### Format

```
HCURSOR WinLoadCursor(hab, idModule, idCursor)  
UINT idModule;  
UINT idCursor;  
HAB hab;
```

### Description

This function loads the cursor resource identified by idCursor from the file associated with the module specified by hInstance. Returns a cursor handle if successful, NULL otherwise.

If idModule, returned by the DOS DosLoadModule, is NULL, the cursor is loaded from the application's resources. Otherwise, idModule is a module handle of a dynamic-link library that contains cursor resources.

### Notes

A new copy of the cursor is created each time WinLoadCursor() is called. The cursor handle must be destroyed with WinDestroyCursor().

To obtain one of the standard system cursors, use GetSysValue(). The standard system cursors must not be freed by the application.

---

## WinCreateCursor

---

### Format

```
HCURSOR WinCreateCursor(hbmCursor, fCursor,  
                        xHotspot, yHotspot)  
HBITMAP hbmCursor;  
BOOL fCursor;  
INT xHotspot, yHotspot;
```

**Description**

This function creates a cursor from the specified bit-map handle and the hotspot values. If `fCursor` is `TRUE`, the bitmap is stretched to have the system cursor dimensions. If `fCursor` is `FALSE`, the bitmap is stretched to have the system icon dimensions.

Returns a cursor handle, or `NULL` if unsuccessful.

---

**WinDestroyCursor**

---

**Format**

```
BOOL WinDestroyCursor (hcsr)
HCURSOR hcsr;
```

**Description**

This function destroys a cursor or icon.

---

**WinSetCursor**

---

**Format**

```
HCURSOR WinSetCursor (hab, hcsr)
HCURSOR hcsr;
HAB hab;
```

**Description**

This function sets the mouse pointer cursor to the specified cursor. Returns the previous cursor handle, or `NULL` if none existed. If `hcsr` is `NULL`, the cursor is removed from the screen.

**Notes**

This function is very fast if `hcsr` is the same as the current cursor handle.

---

**WinShowCursor**

---

**Format**

```
INT WinShowCursor (hab, fShow)
HAB hab;
BOOL fShow;
```

**Description**

This function is used to show or hide the mouse cursor.

Visibility of the mouse cursor is controlled by the mouse cursor level. If the cursor level is  $\leq 0$ , then the mouse cursor is visible; if  $> 0$ , then the cursor is

invisible.

If `fShow` is `TRUE`, then `ShowCursor()` decrements the cursor display level. If the cursor level is decremented to 0, then the mouse cursor is shown.

If `fShow` is `FALSE`, the cursor level is incremented. If it is incremented to 1, the mouse cursor is hidden.

`ShowCursor()` returns the new cursor level.

**Notes** The initial value of the cursor level depends on whether a mouse is installed in the system or not. If a mouse exists, the cursor level is 0 (cursor is visible), otherwise, the cursor level is 1 (cursor is invisible).

To obtain the current cursor display level, use `GetSysValue(SV_CURSORLEVEL)`.

---

## WinSetCursorPos

---

### Format

```
BOOL WinSetCursorPos (hab, x, y)
HAB hab;
INT x, y;
```

### Description

This function sets the position of the mouse cursor to the screen coordinates `x` and `y`. Returns `TRUE` if successful, `FALSE` otherwise

---

## WinGetCursorPos

---

### Format

```
BOOL WinGetCursorPos (hab, lppt)
HAB hab;
LPPPOINT lppt;
```

### Description

This function returns the current mouse pointer position, in screen coordinates, in `*lppt`. The position returned is the true position at the time `GetCursorPos` was called; this function is NOT synchronized with the `WinGetMsg()` and `WinPeekMsg()` functions. Use `WinGetMsgPos()` to get the mouse position of the last message obtained via `WinGetMsg()` or `WinPeekMsg()`. Returns `TRUE` if successful, `FALSE` otherwise.

---

**WinRestrictCursor**

---

**Format**

```
BOOL WinRestrictCursor (hab, lprc)
HAB hab;
LPRECT lprc;
```

**Description**

This function is used to restrict the mouse cursor to the rectangle indicated by lprc. lprc is assumed to point to a rectangle in screen coordinates. If lprc is NULL, then the entire screen is assumed. Returns TRUE if successful, FALSE otherwise.

---

---

**WinQueryCursorInfo**

---

**Format**

```
BOOL WinQueryCursorInfo (hcsr, lpCursorInfo)
HCURSOR hcsr;
CURSORINFO FAR *lpCursorInfo;
```

**Description**

This function is used to obtain a cursor's bitmap handle and hotspot coordinates. This information is placed in \*lpCursorInfo. Returns TRUE if successful, FALSE otherwise.

## **4.1.19 Clipboard Manager**

The Clipboard Manager maintains data to be used for data interchange using the Cut/Copy/Paste editing metaphor. A set of functions is provided to allow applications to easily implement these commands, in such a way that data may be transferred from one application to another.

### **4.1.19.1 Cut/Copy/Paste Editing Metaphor**

The Cut, Copy, and Paste commands work as follows:

---

Command	Description
Copy	Copies the current selection into the clipboard. The previous contents of the clipboard are destroyed.

Cut	Deletes the current selection, after first copying it into the clipboard. The previous contents of the clipboard are destroyed.
Paste	The current selection is deleted, then a copy of the contents of the clipboard is inserted at the selection. The clipboard is not changed.
Clear	Same as Cut, except that the selection is not copied to the clipboard. The clipboard is not changed.

Notice that these commands operate on the contents of the clipboard and the current selection. Logically, there is only one item of data in the clipboard at a time: a selection.

A selection may be placed into the clipboard in a variety of formats. These formats may be defined by the application, or one of the predefined standard clipboard formats may be used. A format is identified by a 16-bit value. Like window messages, format values may be registered dynamically that are guaranteed to be unique across the system.

In order to facilitate transferring data to a wide variety of applications, the selection may be copied to the clipboard in more than one format. When an application obtains data from the clipboard for a Paste command, it can request the data in whatever format is most convenient.

Clipboard data does not need to be generated, or "rendered", at the time it is placed in the clipboard. Instead, the application renders the data at the time another application requests the data for a Paste operation. Because it often takes a considerable amount of time and memory to render a selection in the many formats that an application may be capable of producing, this "delayed rendering" feature is quite useful.

The clipboard is "owned" by the window that last issued the `WinSetclipbrdOwner()` function. There may be data in the clipboard even if the clipboard is unowned, such as when an owner window is destroyed. When clipboard data is requested for a Paste that hasn't been rendered yet, the clipboard owner window is sent a message to render it.

It is not always necessary to own the clipboard when putting data into it. It must be owned in the following circumstances:

- If delayed rendering is used, so that the owner can process the `WM_RENDERFMT` messages
  - If any data with the `CFL_OWNERDISPLAY` flag has been set so that the owner can process the `WM_PAINTCLIPBOARD`, `WM_SIZECLIPBOARD`, `WM_HSCROLLCLIPBOARD`, and `WM_VSCROLLCLIPBOARD` messages
  - If any data with the `CFL_OWNERFREE` flag has been set, so that the owner can process the `WM_DESTROYCLIPBOARD` message
- In any of the cases above, the owner must not terminate until the



WM\_DESTROYCLIPBOARD message has been processed.

In order to access the clipboard, it must be opened. When a thread has opened the clipboard, other threads are not allowed access to the clipboard. Closing the clipboard allows other threads to access the clipboard. Once data has been set into the clipboard, the program that set the data should not change it.

Generally, clipboard operations should be performed only as a direct or indirect result of some user operation.

It is possible for an application to register a window as a "Clipboard Viewer" window. This window is responsible for displaying the contents of the clipboard. It is notified of changes to the clipboard by being sent certain window messages when the clipboard changes. Only one window at a time may be registered as the clipboard viewer.

The clipboard viewer can display any of the standard clipboard formats. However, since the clipboard viewer cannot display data that has not been rendered yet, there are a few special predefined clipboard formats that are used for clipboard viewer display purposes only. If any of these special formats exist in the clipboard, the clipboard viewer will display these formats rather than attempt to display the others. These are the CF\_DSPTEXT and CF\_DSPBITMAP formats defined below.

To display private clipboard formats, the clipboard viewer application may send messages to the clipboard owner window to draw the clipboard.

#### **4.1.19.2 Standard Clipboard Formats**

---

Standard Clipboard Formats  
Format

##### **CF\_TEXT**

Text format. Each line ends with a carriage return/linefeed (CR-LF) combination. Tab characters separate fields within a line. A NULL character signals the end of the data.

##### **CF\_BITMAP**

Bitmap as defined by the BITMAP data structure.

##### **CF\_DSPTEXT**

Text display format associated with private format.

##### **CF\_METAFILE**

Metafile format, as defined in the GPI metafile section. The clipboard viewer will display this format in preference to privately formatted data.

## CF\_DSPBITMAP

Bitmap display format associated with private format. The clipboard viewer will display this format in preference to privately formatted data.

In addition to the above predefined formats, any format value registered through the standard system atom manager can be used as the `fmt` parameter.

### 4.1.19.3 Clipboard Functions

---

#### WinOpenClipbrd

---

##### Format

```
BOOL WinOpenClipbrd(hab)
HAB hab;
```

##### Description

This function opens the clipboard for use, preventing other threads or processes from examining or changing the clipboard contents. If another thread or process has the clipboard open, this function does not return until it is closed. Returns TRUE if clipboard was successfully opened.

Messages may be received from other threads or processes.

---

#### WinCloseClipbrd

---

##### Format

```
BOOL WinCloseClipbrd(hab)
HAB hab;
```

##### Description

This function closes the clipboard, allowing other applications to open the clipboard with `WinOpenClipbrd()`. Returns TRUE if successful, FALSE otherwise.

---

#### WinEmptyClipbrd

---

##### Format

```
BOOL WinEmptyClipbrd(hab)
HAB hab;
```

**Description**

This function empties the clipboard and frees handles to data in the clipboard. Returns TRUE if successful, FALSE otherwise

---

**WinSetClipbrdOwner**

---

**Format**

```
WinSetClipbrdOwner (hab, hwnd)
HWND hwnd;
HAB hab;
```

**Description**

This function sets the current clipboard owner window to hwnd. The owner window receives the WM\_RENDERFMT, WM\_DESTROYCLIPBOARD, and WM\_SIZECLIPBOARD, WM\_VSCROLLCLIPBOARD, WM\_HSCROLLCLIPBOARD, and WM\_PAINTCLIPBOARD messages at the appropriate times.

---

**WinQueryClipbrdOwner**

---

**Format**

```
HWND WinQueryClipbrdOwner (hab, fLock)
HAB hab;
BOOL fLock;
```

**Description**

This function returns the window handle of the current clipboard owner, or NULL if the clipboard is not owned. If fLock is TRUE, then the window handle is locked before returning.

---

**WinSetClipbrdData**

---

**Format**

```
HANDLE WinSetClipbrdData (h, fmt, rgfFmtInfo)
HANDLE h;
UINT fmt;
UINT rgfFmtInfo;
```

**Description**

This function sets data of the specified format into the clipboard. h is a handle to a data object of the format specified by fmt. If h is NULL, then the clipboard

owner window will be sent a `WM_RENDERFMT` message to render the format if `WinQueryClipbrdData()` is called with the specified format.

`rgfFmtInfo` contains one of the following flags that describe the type of the data handle:

---

**CFL\_SELECTOR**

Handle is a segment, freed with `DosFreeSeg()`. The segment must be allocated shareable

**CFL\_OWNERFREE**

Handle is not freed by `WinEmptyClipboard()`. The application must free the data if necessary.

**CFL\_OWNERDISPLAY**

This flag indicates that the format will be drawn by the clipboard owner in the clipboard viewer window via the `WM_PAINTCLIPBOARD` message. The `h` parameter should be `NULL`.

Note that if there is previous data in the clipboard, then it is freed by this call.

---

---

## WinQueryClipbrdData

---

**Format**

```
HANDLE WinQueryClipbrdData(hab, fmt)
HAB hab;
UINT fmt;
```

**Description**

This function returns the clipboard data handle of the format indicated by `fmt`, or `NULL` if that format does not exist in the clipboard.

**Notes**

The returned data handle must not be accessed after `WinCloseClipbrd()` is called. For this reason, the application must either copy the data for long term use or process the data before `WinCloseClipbrd()` is called. The application should not free the data handle or leave it locked.

---

---

## WinEnumClipbrdFmts

---

**Format**

```

UINT WinEnumClipbrdFmts (hab, fmtPrev)
HAB hab;
UINT fmtPrev;

```

**Description**

This function is used to enumerate all the formats available in the clipboard. If `fmtPrev` is 0, then the first available format is returned. Otherwise, `fmtPrev` should be set to the last format returned, in which case `WinEnumClipbrdFmts()` will return the next available format. Enumeration is complete when `WinEnumClipbrdFmts()` returns 0.

**Example**

The example below counts the number of available formats, and fills an array with the format values:

```

fmt = 0;
cFmts = 0;
while ((fmt = WinEnumClipbrdFmts(fmt)) != 0) {
    rgfmt[cFmts++] = fmt;
}

```

---

**WinQueryClipbrdFmtInfo**

---

**Format**

```

BOOL WinIsClipbrdFmtAvail (hab, fmt, lprgfFmtInfo)
HAB hab;
UINT fmt;
UINT FAR *lprgfFmtInfo;

```

**Description**

This function returns TRUE if data of the format indicated by `fmt` is available, FALSE otherwise. This function does NOT cause the data to be rendered. If TRUE is returned, the format information (CFL\* flags) is returned in `*lprgfFmtInfo`.

**4.1.19.4 Clipboard messages**

---

**WM\_RENDERFMT**

---

**Format**

```

WM_RENDERFMT
LOUINT(lParam1):    UINT fmt;
lParam2:            OL

```

Returns: OL

Description

This message is a request to the clipboard owner to render the data of the format specified in lParam1. The data should be rendered into a global handle, which should then be set into the clipboard with WinSetClipbrdData().

---

## WM\_RENDERALLFMTS

---

Format

WM\_RENDERALLFMTS  
lParam1: O  
lParam2: OL  
Returns: O

Description

This message is sent to the application that owns the clipboard when the application is being destroyed. The application should render the clipboard data in all formats it is capable of generating and pass a handle to each format to WinSetClipbrdData. This ensures that the data in the clipboard can be rendered even though the application has been destroyed.

---

## WM\_DESTROYCLIPBOARD

---

Format

WM\_DESTROYCLIPBOARD  
lParam1: O  
lParam2: OL  
Returns: O

Description

This message is sent to the clipboard owner when the clipboard is emptied through a call to WinEmptyClipbrd(). If there is any data that has been set with the CFL\_OWNERFREE flag, the clipboard owner must free the data at this time.

---

## WM\_PAINTCLIPBOARD

---

Format

WM\_PAINTCLIPBOARD  
lParam1: HWND hwndViewer;  
lParam2: PAINTSTRUCT FAR \*lpPaintStruct;

Returns: OL;

Description

This message is sent when the clipboard contains a data handle with the `CFI_ OWNERDISPLAY` information flag set (i.e., the clipboard owner is responsible for displaying the clipboard contents) and the clipboard application's client area needs repainting. The `WM_PAINTCLIPBOARD` message is sent to the owner of the clipboard to request repainting of all or part of the clipboard application's client area.

`lParam1` is a handle to the clipboard application window.

`lParam2` is a long pointer to a `PAINTSTRUCT` data structure defining what part of the client area to paint.

Notes

To determine whether the entire client area needs repainting or just a portion of it, the clipboard owner must compare the dimensions of the drawing area given in the `repaint` field of the `PAINTSTRUCT` structure to the dimensions given in the most recent `WM_SIZECLIPBOARD` message.

---

## WM\_SIZECLIPBOARD

---

Format

```
WM_SIZECLIPBOARD
lParam1:  HWND hwndViewer;
lParam2:  RECT FAR *lprcPaint;
Returns:  OL
```

Description

This message is sent when the clipboard contains a data handle for the `CF_ OWNERDISPLAY` format (i.e., the clipboard owner is responsible for displaying the clipboard contents) and the clipboard application window has changed size.

`lParam1` is a handle to the clipboard application window. `lParam2` is a pointer to a `RECT` data structure specifying the area in which the clipboard owner should paint.

Notes

A `WM_SIZECLIPBOARD` message is sent with a pointer to an empty rectangle (0,0, 0,0) as the new size when the clipboard application is about to be destroyed or made iconic. This permits the clipboard owner to free its display resources.

---

**WM\_HSCROLLCLIPBOARD**

---

**Format**

```
WM_HSCROLLCLIPBOARD
lParam1:          HWND hwndViewer;
LOUINT(lParam2):  INT posScroll;
HIUINT(lParam2):  INT codeScroll;
Returns: OL
```

---

---

**WM\_VSCROLLCLIPBOARD**

---

**Format**

```
WM_VSCROLLCLIPBOARD
lParam1:          HWND hwndViewer;
LOUINT(lParam2):  INT posScroll;
HIUINT(lParam2):  INT codeScroll;
Returns: OL
```

**Description**

These messages are sent to the clipboard owner window when the clipboard contains a data handle for the CF\_OWNERDISPLAY format (i.e., the clipboard owner is responsible for displaying the clipboard contents) and an event occurs in the clipboard application's horizontal scroll bar.

lParam1 contains a handle to the clipboard application window.

HIUINT(lParam2) contains one of the SB\_\* scroll bar codes as defined in the "Scroll Bar Controls" section.

Except for SB\_THUMBPOSITION, LOUINT(lParam2) contains 0.

**Notes**

The clipboard owner should use WinInvalidateRect or repaint as desired. The scroll bar position should also be reset.

---

**4.1.19.5 Clipboard Viewer Functions**

---

---

**WinSetClipbrdViewer**

---

**Format**

```
HWND WinSetClipbrdViewer(hab, hwndViewer)
HWND hwndViewer;
HAB hab;
```



**Description**

This function sets the current clipboard viewer window to `hwndViewer`. It returns the previous clipboard viewer, or `NULL` if there was none. The returned window handle is not locked.

---

**WinQueryClipbrdViewer**

---

**Format**

```
HWND WinQueryClipbrdViewer (hab, fLock)
BOOL fLock;
HAB hab;
```

**Description**

This function returns the current Clipboard viewer window or `NULL` if there isn't one. If `fLock` is `TRUE`, the window is locked before returning. If `fLock` is `FALSE`, the window is not locked by this call (it may already have been locked by some other call).

---

**WM\_DRAWCLIPBOARD**

---

**Format**

```
WM_DRAWCLIPBOARD
lParam1:    OL
lParam2:    OL
Returns:    OL
```

**Description**

This message is sent to the clipboard viewer window whenever the contents of the clipboard change.

**4.1.19.6 Clipboard Examples****4.1.19.6.1 Cutting or Copying; no delayed rendering**

1. Open the clipboard with `WinOpenClipbrd()`.
2. Call `WinEmptyClipbrd()` to delete the previous contents of the clipboard.
3. Using the current selection, create clipboard data handle in the desired format(s) and place each handle into the clipboard with `WinSetClipbrdData()`. Once placed in the clipboard, the handles are no longer valid, and must not be used again by the application.

4. After all of the desired formats have been added to the clipboard, close the clipboard with `WinCloseClipbrd()`.

#### *4.1.19.6.2 Cutting or Copying with delayed rendering*

1. Open the clipboard with `WinOpenClipbrd()`.
2. Call `WinEmptyClipbrd()` to delete the previous contents of the clipboard
3. Call `WinSetClipbrdOwner()` to set the ownership of the clipboard
4. Call `WinSetClipbrdData()` with the desired format values and a NULL data handle.
5. If desired, create data in any of the standard display formats, and put the handles in the clipboard with `WinSetClipbrdData()`.
6. Close the clipboard.
7. When a `WM_RENDERFORMAT` message is recieved by the clipboard owner window, the data of the requested format should be rendered and placed into the window with `WinSetClipbrdData()`. It is not necessary to open the clipboard when processing the `WM_RENDERFORMAT` message.
8. If a `WM_PAINTCLIPBOARD`, `WM_SIZECLIPBOARD`, `WM_HSCROLLCLIPBOARD` or `WM_VSCROLLCLIPBOARD` mesaged is receive, carry out the requested function to repaint the clipboard.
9. If a `WM_DESTROYCLIPBOARD` message is received, free up any data in the clipboard.

#### *4.1.19.6.3 Pasting*

1. Open the clipboard with `WinOpenClipbrd()`.
2. Call `WinQueryClipbrdData()` with the format that is most conveniently handled by the application doing the Paste. If NULL is returned to indicate that data in that format is not available, try any other formats that the application knows how to Paste.
3. If a handle is obtained with `WinQueryClipbrdData()`, it may be used as long as the clipboard remains open. Once the clipboard is closed, the handle may be deleted. Since leaving the clipboard open for long periods of time may prevent other applications from accessing the clipboard, it's a good idea to make a copy of the clipboard data, and then close the clipboard.
4. When finished with the clipboard data handle, close the clipboard with `WinCloseClipbrd()`.

## 4.1.20 Rectangle Functions

### 4.1.20.1 Data structures

This section documents the rectangle and point utility functions. The rectangle structure is:

```
typedef struct {
    INT xLeft;
    INT dummy1;
    INT yBottom;
    INT dummy2;
    INT xRight;
    INT dummy3;
    INT yTop;
    INT dummy4;
} RECT;
```

```
typedef RECT FAR *LRECT;
```

The point structure is:

```
typedef struct {
    INT x;
    INT dummy1;
    INT y;
    INT dummy2;
} POINT
```

A POINT structure describes a window coordinate point. A RECT structure describes a rectangular area in the window coordinate space: it is stored as an array of 2 points, the first point being the bottom left corner of the rectangle, and the second point being the top right corner.

An empty rectangle is a rectangle that has no area: the right coordinate is less than or equal to the left, top is less than or equal to the bottom.

The rectangle routines assume that coordinates increase as you go up or to the right.

Logically speaking, the right and top coordinates of a rectangle are one greater than the last coordinate included in the rectangle. Also, some of the rectangle routines assume that coordinates used are in window coordinates, where left and bottom most coordinates are smaller than the right or top coordinates.

To calculate the dimensions in pixels:

```
cy = rc.yTop - rc.yBottom
cx = rc.xRight - rc.xLeft;
```

To determine whether a coordinate falls inside a rectangle:

```
fInside = ( (rc.xLeft <= x && x < rc.xRight)
            && (rc.yBottom <= y && y < rc.yTop) );
```

Note that the comparisons to right and bottom are done with "less than".

When passed to a WinXXX function, the values of dummy1..3 are not significant. The system may alter the values of these fields, however.

These structures are very similar to the GRECT and GPOINT structures, except that only the lower 16 bits of the values are significant. A RECT or POINT can be converted to a GRECT or GPOINT by simply sign extending the 16 bit fields into the hi-order 16 bits.

The functions WinMakeGRect and WinMakeGPoint convert RECT and POINT structures to GRECTs and GPOINTs respectively, by sign extending the significant fields into the dummy fields.

Any function that returns a rectangle or point (WinGetWindowRect, WinBeginPaint, etc.) always sign extends the significant fields into the dummy fields; that is, the returned structures can be used as GRECTs or GPOINTs.

#### 4.1.20.2 Rectangle routines

---

##### WinSetRect

---

###### Format

```
void WinSetRect(hab, lprc, left, bottom,
               right, top)
LPRECT lprc;
INT left;
INT top;
INT right;
INT bottom;
HAB hab;
```

###### Description

This function fills the rectangle pointed to by lprc with the passed coordinates. This routine is equivalent to assigning the left, top, right, and bottom arguments to the appropriate fields of \*lprc.

---

##### WinIsRectEmpty

---

###### Format

```
BOOL WinIsRectEmpty(hab, lprc)
LPRECT lprc;
HAB hab;
```

**Description**

This function returns TRUE if \*lprc is an empty rectangle, FALSE otherwise. An empty rectangle is one that has no area: right is less than or equal to left, bottom is less than or equal to top.

**Notes**

This function works only if the top and left coordinates of \*lprc are less than or equal to the bottom and right coordinates.

---

**WinCopyRect**

---

**Format**

```
void WinCopyRect(hab, lprcDest, lprcSrc)
LPRECT lprcDest;
LPRECT lprcSrc;
HAB hab;
```

**Description**

This function copies the rectangle from \*lprcSrc to \*lprcDest.

---

**WinEqualRect**

---

**Format**

```
BOOL WinEqualRect(hab, lprc1, lprc2)
LPRECT lprc1;
LPRECT lprc2;
HAB hab;
```

**Description**

This function returns TRUE if \*lprc1 and \*lprc2 are identical, FALSE otherwise.

---

**WinSetRectEmpty**

---

**Format**

```
void WinSetRectEmpty(hab, lprc)
LPRECT lprc;
HAB hab;
```

**Description**

This function sets \*lprc to an empty rectangle by setting each field to 0. Equivalent to WinSetRect(lprc, 0, 0, 0).

---

## WinOffsetRect

---

### Format

```
void WinOffsetRect(hab, lprc, cx, cy)
LPRECT lprc;
INT cx;
INT cy;
HAB hab;
```

### Description

This function offsets the coordinates of \*lprc by adding cx to both the left and right coordinates, and cy to both the top and bottom coordinates.

---

## WinInflateRect

---

### Format

```
VOID WinInflateRect(hab, lprc, cx, cy)
LPRECT lprc;
INT cx;
INT cy;
HAB hab;
```

### Description

This function expands the given rectangle by cx horizontally and cy vertically on all sides. If cx or cy is negative, the rectangle is inset. cx is subtracted from the left and added to the right, and cy is subtracted from the bottom and added to the top.

---

## WinPtInRect

---

### Format

```
BOOL WinPtInRect(hab, lprc, pt)
LPRECT lprc;
POINT pt;
HAB hab;
```

### Description

This function returns TRUE if pt falls inside of \*lprc.

**Notes** This function works only if the bottom and left coordinates of \*lprc are less than or equal to the top and right coordinates.

---

**WinIntersectRect**

---

**Format**

```
BOOL WinIntersectRect(hab, lprcDest,  
                      lprcSrc1, lprcSrc2)  
LPRECT lprcDest;  
LPRECT lprcSrc1;  
LPRECT lprcSrc2;  
HAB hab;
```

**Description**

Calculates the intersection between \*lprcSrc1 and \*lprcSrc2, returning the resulting rectangle in \*lprcDest. Returns TRUE if \*lprcSrc1 intersects \*lprcSrc2, FALSE otherwise. If there is no intersection, an empty rectangle is returned in \*lprcDest.

**Notes**

This function works only if the bottom and left coordinates of both \*lprcSrc1 and \*lprcSrc2 are less than or equal to the top and right coordinates .

---

---

**WinUnionRect**

---

**Format**

```
BOOL WinUnionRect(hab, lprcDest,  
                  lprcSrc1, lprcSrc2)  
LPRECT lprcDest;  
LPRECT lprcSrc1;  
LPRECT lprcSrc2;  
HAB hab;
```

**Description**

This function calculates a rectangle that bounds \*lprcSrc1 and \*lprcSrc2, returning the result in \*lprcDest. If either \*lprcSrc1 or \*lprcSrc2 are NULL, then the other rectangle is returned. Returns TRUE if \*lprcDest is a non-empty rectangle, FALSE otherwise.

**Notes**

This function works only if the bottom and left coordinates of both \*lprcSrc1 and \*lprcSrc2 are less than or equal to the top and right coordinates .

---

---

**WinSubtractRect**

---

**Format**

```
BOOL WinSubtractRect(hab, lprcDest,  
                     lprc1, lprc2)  
LPRECT lprcDest;
```

```
LPRECT lprcSrc1;  
LPRECT lprcSrc2;  
HAB hab;
```

Description

This function subtracts \*lprc2 from \*lprc1, returning the result in \*lprcDest. Returns TRUE if \*lprcDest is empty, FALSE otherwise.

Notes

Subtracting one rectangle from another may not always result in a rectangular area; in this case SubtractRect will return \*lprc1 in \*lprcDest. For this reason, SubtractRect provides only an approximation of subtraction. However, the area described by \*lprcDest will always be greater than or equal to the "true" result of the subtraction.

You can use the GPICombineRgn() function to calculate the true result of the subtraction of two rectangular areas. SubtractRect() is much faster, however.

This function works only if the bottom and left coordinates of both \*lprcSrc1 and \*lprcSrc2 are less than or equal to the top and right coordinates .

---

## WinMakeGRect

---

Format

```
BOOL WinMakeGRect(lprc)  
LPRECT lprc;
```

Description

This function converts the RECT structure referenced by lprc into a GRECT structure.

Returns TRUE if the conversion is successful, FALSE otherwise.

---

## WinMakeGPoint

---

Format

```
BOOL WinMakeGPoint(lppt)  
LPPOINT lppt;
```

Description

This function converts the POINT structure referenced by lprc into a GPOINT structure.

Returns TRUE if the conversion is successful, FALSE otherwise.



### 4.1.21 Presentation Manager Resources

---

#### WinLoadString

---

##### Format

```
int WinLoadString(hab, idModule, idString,
                  lpszBuffer, cchBufferMax)
HAB hab;
UINT idModule;
UINT idString;
LPSTR lpszBuffer;
int cchBufferMax;
```

##### Description

This function loads a string resource identified by `idString` from the executable file associated with module `idModule` which is returned by the DOS `DosLoadModule` call. The function copies the string into the buffer pointed to by `lpBuffer`, and appends a terminating null character. Returns the size of the string copied into `lpszBuffer`, which is no larger than `(cchBufferMax - 1)`.

If `idModule` is `NULL`, a bitmap from the application resource file is loaded. Otherwise, `idModule` is the module handle of a dynlink library containing the bitmap resource.

For loading bitmaps, see `WinLoadBitmap`.

### 4.1.22 Command Key Accelerators

*Accelerators* are keyboard keystrokes which can be used to invoke commands directly. The normal way in which this is done is to convert the keystrokes into `WM_COMMAND`, `WM_SYSCOMMAND` or `WM_HELP` messages before they are received by the application, during the `WinGetMsg` or `WinPeekMsg` functions. This can allow single keystrokes to appear the same as selecting an item off a menu, to the user and the application alike.

The Accelerator Table is used to define which keystrokes are treated as accelerators and the Commands they are translated into.

The `WinSet/GetAccelTable()` functions are used to set and get the accelerator table that is used implicitly by the `WinGetMsg()` and `WinPeekMsg()` functions. A `WM_SYSCOMMAND`, `WM_COMMAND` or `WM_HELP` message is sent, depending on whether or not the `SYSCOMMAND` or `HELP` flags are set in the accelerator option definition.

Note that the Accelerator tables are used in a two-level fashion. There is a table which operates on a per-input-queue basis and also one which works on a per-active-window basis. Both are used in succession during message processing in `WinGetMsg` and `WinPeekMsg`. This two level structure allows for some accelerators which are global across all windows (F1=Help, for example) as well as accelerators which are closely allied to the menu of a particular window.

Translation within `WinGetMsg` is done with the active window's accelerator table first and subsequently with the queue table. This implies that entries in the queue table can be overridden by entries in the active window table.

The following is the structure of an accelerator table:

```
typedef struct tagACCELTABLE {
    UINT cAccel;
    UINT codepage;
    ACCEL rgaccel(cEntries);
} ACCELTABLE;
```

- `cAccel` is the count of accelerators in the table.
- `codepage` is the code page assumed by all `AF_CHAR` entries (only code points are tied to a particular code page, scan codes and virtual keys are not).

```
typedef struct tagACCEL {
    UINT rgf;
    UINT key;
    UINT cmd;
} ACCEL;
```

- `rgf` contains a combination of the following flags:

#### `AF_CHAR`

key contains a codepoint (translated character) (codepoint value is in codepage identified in `ACCELTABLE` structure above).

#### `AF_VIRTUALKEY`

key contains a virtual key value

#### `AF_SCANCODE`

key contains a scan code value

NOTE - the three values above have the same value as their `KC_*` counterparts.

#### `AF_SHIFT`

Shift key must be down

#### `AF_CONTROL`

Control key must be down

**AF\_ALT**

Alt key must be down

**AF\_SYSCOMMAND**

Produce WM\_SYSCOMMAND instead of WM\_COMMAND

**AF\_HELP**

Produce WM\_HELP instead of WM\_COMMAND

- key contains the virtual key, scan code, or code point value (depending on the contents of rgf).
- cmd contains the command ID value to be placed in lParam1 of the resultant WM\_COMMAND, WM\_SYSCOMMAND, or WM\_HELP message.

```
typedef {  
    ULONG    HACCEL;  
}
```

---

**WinLoadAccelTable**

---

**Format**

```
HACCEL WinLoadAccelTable(hab, idModule, idAccelTable)  
HAB  
UINT idModule;  
UINT idAccelTable;
```

**Description**

Loads an accelerator table identified by idModule, which is returned by the DOS DosLoadModule call, and idAccelTable, returning the handle of the accelerator table.

Note that this function will return a different hAccel value when called twice in succession with the same parameter values.

---

**WinCreateAccelTable**

---

**Format**

```
HACCEL haccel = WinCreateAccelTable(hab, lpAccel)  
HAB hab;  
ACCELTABLE FAR *lpAccel;
```

**Description**

Given a pointer to an accelerator table in memory, lpAccel, this function creates an accelerator table handle. lpAccel points to an ACCELTABLE structure.

Note that this function will return a different hAccel

value when called twice in succession with the same parameter values.

---

### WinDestroyAccelTable

---

#### Format

```
BOOL fDestroyed = WinDestroyAccelTable(haccel)  
HACCEL haccel;
```

#### Description

This function destroys an accelerator table, returning TRUE if successful, FALSE otherwise.

---

### WinCopyAccelTable

---

#### Format

```
WORD cbCopied = WinCopyAccelTable(haccel, lpAccel,  
                                cbCopyMax)  
HACCEL haccel;  
ACCELTABLE FAR *lpAccel;  
WORD cbCopyMax;
```

#### Description

This function is used to either obtain the accelerator table data corresponding to an accel table handle, or to determine the size of the table data.

If the lpAccel pointer is not NULL, then up to cbCopyMax bytes of the accelerator table data is copied to lpAccel. The actual number of bytes copied is returned.

If lpAccel is NULL, then this function returns the size in bytes of the accelerator table handle; the cbCopyMax parameter is ignored.

---

### WinTranslateAccel

---

#### Format

```
BOOL WinTranslateAccel(hab, hwnd, hAccel, lpqmsg)  
HAB hab;  
HWND hwnd;  
HACCEL hAccel;  
QMSG FAR *lpQmsg;
```

#### Description

This function translates the message pointed to by lpqmsg if it is a WM\_CHAR message that is in the

accelerator table indicated by `hAccel`. The message is translated into a `WM_COMMAND`, `WM_SYSCOMMAND` or `WM_HELP` message, with `hwnd` identifying the destination window. Normally, this parameter should be a frame window handle. This function does not hilite menu items.

If `hAccel` is `NULL`, the current accelerator table is assumed.

`WinTranslateAccel()` returns `TRUE` if the message matched an accelerator in the table. The message pointed to by `lpqmsg` is modified by `WinTranslateAccel()` if a match is found.

If a menu item exists that matches the accelerator command value, and that item is disabled, the message at `*lpqmsg` is translated to a `WM_NULL` message, rather than a `WM_COMMAND`, `WM_SYSCOMMAND` or `WM_HELP` message. If the command is a `WM_COMMAND` or `WM_HELP`, the menu child window of `hwnd` that has the `FID_MENU` ID is searched; if `WM_SYSCOMMAND`, the `FID_SYSMENU` child window is searched.

It is possible to have accelerators that do not correspond to items in a menu. If the command value does not match any items in the menu, the message is `STILL` translated.

Generally, applications do not have to call this function. It is normally called automatically by `WinGetMsg()` and `WinPeekMsg()`, when a `WM_CHAR` message is received, with the window handle of the active window as the first parameter. The standard frame window procedure always passes `WM_COMMAND` messages to the `FID_CLIENT` window. Since the message is physically changed by `WinTranslateAccel()`, this implies that applications will not see the `WM_CHAR` messages that resulted in `WM_COMMAND`, `WM_SYSCOMMAND` or `WM_HELP` messages.

---

## WinSetAccelTable

---

### Format

```
BOOL FAR PASCAL WinSetAccelTable(hab, haccel,
                                hwndFrame)
HAB hab;
HACCEL haccel;
HWND hwndFrame;
```

**Description**

This function is used to set either the window or queue accelerator table. If `hwndFrame` is `NULL`, then the queue accelerator table is set. Otherwise, the window accelerator table is set, by sending the `WM_SETACCELTABLE` message to `hwndFrame`.

If `hAccel` is `NULL`, the effect of this function is to remove any accelerator table in effect for the window or queue.

---

**WinQueryAccelTable**

---

**Format**

```
HACCEL FAR PASCAL WinQueryAccelTable(hab, hwndFrame)
HAB hab;
HWND hwndFrame;
```

**Description**

This function is used to query either the window or queue accelerator table. If `hwndFrame` is `NULL`, then the queue accel table is returned. Otherwise, the window accel table is returned, by sending the `WM_QUERYACCELTABLE` message to `hwndFrame`.

**4.1.22.1 Accelerator Table Messages.**

The following messages are associated with Accelerator Table functions:

---

**WM\_SETACCELTABLE**

---

**Format**

```
WM_SETACCELTABLE
lParam1:    HACCEL  haccelNew
Returns:    BOOL  fSuccess;
```

**Description**

`WM_SETACCELTABLE` is used to establish the window accelerator table to be used for translation when the window is active.

---

**WM\_QUERYACCELTABLE**

---

**Format**

```
WM_QUERYACCELTABLE
Returns:    HACCEL  haccel;
```

**Description**

WM\_QUERYACCELTABLE returns the accel table handle associated with the window, or NULL if none is associated.

---

**WM\_VALIDATEACCEL**

---

**Format**

```
WM_VALIDATEACCEL
lParam1:    LPQMSG lpqmsg;
lParam2:    OL;
Returns:    BOOL fValid;
```

**Description**

This message is sent by WinTranslateAccelerator() to the the window that will be the destination of the WM\_COMMAND/SYSCOMMAND/HELP message when a match is found in the accelerator table.

lParam1 points to a QMSG structure containing WM\_COMMAND/SYSCOMMAND/HELP message that will be returned. The message should return FALSE to indicate that the message is invalid and should not be passed on to the application, and TRUE to indicate that the message should be passed on to the app.

This message is processed by the WC\_FRAME and WC\_DIALOG window procedures by seeing if the command exists in either the system or application menu (FID\_SYSMENU and FID\_MENU frame controls). If so, if the corresponding menu item is disabled, FALSE is returned. If the item is enabled, TRUE is returned. If the command does not exist in either menu, TRUE is returned.

These messages are processed only by frame windows.

**4.1.22.2 Default Queue Accelerator Table.**

The default queue accelerator table contains entries for the following functions:

- Enter menu mode (F2)
- Bring up system menu
- Help (F1)

Switch application and switchlist keys are not part of this table. These are fixed by OS/2 and cannot be overridden in the accelerator tables.

### 4.1.23 System Colors

The System Colors are the colors that the system and applications use to color various parts of the user interface. These colors may be modified by the application

The system color functions are used to get and change the system colors. The

#### 4.1.23.1 System Color Index Values

---

System Color Indices

Color Index Value

SCLR\_SCROLLBAR

Scroll bar halftone area

SCLR\_BACKGROUND

Desktop

SCLR\_ACTIVECAPTION

Active window caption

SCLR\_INACTIVECAPTION

Inactive window caption

SCLR\_MENU

Menu background

SCLR\_WINDOW

Window background and thumb box

SCLR\_WINDOWFRAME

Window border, caption text background

SCLR\_MENUTEXT

Text in menus

SCLR\_WINDOWTEXT

Text in windows

SCLR\_TITLETEXT

Text in title bar, size box, scroll bar arrow box



### 4.1.23.2 System Color Routines

---

#### WinQuerySysColor

---

##### Format

```
ULONG WinQuerySysColor(hab, iColor)
HAB hab;
int iColor;
```

##### Description

This function returns the RGB color value that corresponds to the system color index indicated by iColor. iColor must be one of the SCLR\_\* constants in the table below.

---

#### WinSetSysColors

---

##### Format

```
void WinSetSysColors(hab, cColors, rgiColors,
                    rglColorValues)
HAB hab;
int cColors;
int far *rgiColors;
ULONG far *rglColorValues;
```

##### Description

This functions changes one or more of the system colors. cColors is the count of the number of colors that are being changed, rgiColors is a far pointer to an array of system color indices, and rglColorValues is a far pointer to an array of ULONGs that represent RGB color values. Values in the rgiColors array must be SCLR\_\* values from the table below.

##### Notes

WinSetSysColors() sends all top-level windows in the system a WM\_SYSCOLORCHANGE message to indicate that the colors have changed. When this message is received, applications that depend on the system colors can query the new color values with WinGetSysColor().

After the WM\_SYSCOLORCHANGE messages are sent, all windows in the system are invalidated so that they will be redrawn with the new system colors.

WinSetSysColors() does NOT write any system color changes to the WIN.INI file.

### 4.1.23.3 System Color Messages

---

#### WM\_SYSCOLORCHANGE

---

##### Format

```
WM_SYSCOLORCHANGE
lParam1:          0
lParam2:          OL
Returns:          OL
```

##### Description

This message is sent to all top level windows when a change is made to the system colors with `WinSetSysColors()`. When this message is received, applications that depend on the system colors can query the new color values with `WinGetSysColor()`.

After the `WM_SYSCOLORCHANGE` messages are sent, all windows in the system are invalidated so that they will be redrawn with the new system colors.

### 4.1.24 System Information Functions

This section describes functions used to retrieve and change system characteristics.

The system value functions are used to get and change system information such as the height and width of the screen, dimensions of the various parts of a window, caret blink rate, etc.

---

#### WinGetSysValue

---

##### Format

```
int WinGetSysValue(hwndDesktop, iSysVal)
HWND hwndDesktop;
int iSysVal;
```

##### Description

`WinGetSysValue` returns the system value indicated by `iSysVal`. `hwndDesktop` is the desktop window handle. `iSysVal` must be one of the `SV_*` constants in the table below.

##### Notes

`NULL` may be specified to obtain the system values for the screen device.

---

**WinSetSysValue**

---

**Format**

```
BOOL WinSetSysValue(hwndDesktop, iSysVal, value)
HAB hab;
int iSysVal;
int value;
HWND hwndDesktop;
```

**Description**

WinSetSysValue sets a new system value indicated by iSysVal to value, returning TRUE if successful, FALSE otherwise. iSysVal must be one of the SV\_\* constants in the table below. hwndDesktop is the desktop window handle.

**Notes**

Not all SV\_ values can be set with WinSetSysValue. NULL may be specified to obtain the system values for the screen device.

**4.1.24.1 System Value Constants**

Below is a table of the available system values, which can be used with the WinGetSysValue and WinSetSysValue functions. Dimension values are in pixel units, and time values are in milliseconds. Note that not all system values are settable with WinSetSysValue; those that are changeable are marked with an "\*".

---

WinGetSysValue()	Codes	Settable?	Meaning
------------------	-------	-----------	---------

SV_CXSCREEN			Width of screen
-------------	--	--	-----------------

SV_CYSCREEN			Height of screen
-------------	--	--	------------------

SV_CXVSCROLL			Vertical scroll bar width
--------------	--	--	---------------------------

SV_CYHSCROLL			Horizontal scroll bar height
--------------	--	--	------------------------------

SV_CYVSCROLLARROW			Height of vertical scroll bar arrow bitmaps
-------------------	--	--	---

SV_CXHSCROLLARROW			Width of horizontal scroll bar arrow bitmaps
-------------------	--	--	--

SV_CYCAPTION	Height of caption
SV_CXBORDER	Width of nominal-width border
SV_CYBORDER	Height of nominal-width border
SV_CXSIZEBORDER	Width of sizing border
SV_CYSIZEBORDER	Height of sizing border
SV_CXDLGFRAME	Width of dialog frame border
SV_CYDLGFRAME	Height of dialog frame border
SV_CYVTHUMB	Height of vertical scroll bar thumb
SV_CXHTHUMB	Width of horizontal scroll bar thumb
SV_CXMINMAXBUTTON	Width of Minimize/Maximize buttons
SV_CYMINMAXBUTTON	Height of Minimize/Maximize buttons
SV_CXSIZEBUTTON	Width of size buttons
SV_CYSIZEBUTTON	Height of size buttons
SV_CYMENU	Single line menu height
SV_CXFULLSCREEN	Client area width when window is full screen
SV_CYFULLSCREEN	Client area height when full screen (excluding menu height)
SV_CXICON	Icon width
SV_CYICON	Icon height
SV_CXCURSOR	Cursor width

**SV\_CYCURSOR**  
Cursor height

**SV\_DEBUG**  
TRUE if debugging version of system, FALSE otherwise

**SV\_MOUSEPRESENT**  
TRUE if system has mouse installed, FALSE otherwise

**SV\_FULLSCREEN**  
TRUE if full screen window is present, FALSE otherwise

**SV\_CURSORLEVEL**  
Cursor hide level (0 ==> visible)

**SV\_CTIMERS**  
Count of available timers

**SV\_SWAPBUTTON** \*  
TRUE if mouse buttons swapped. Normally, the mouse buttons are set for right-handed use. Setting this value changes them around for left-handed usage.  
  
If TRUE, **WM\_LBUTTONDOWN\*** messages are returned when the user presses the right button, and **WM\_RBUTTONDOWN\*** messages are returned when the left button is pressed. Modifying this value affects the entire system. Applications should not normally read or set this value. The user normally updates this value via the User Interface Shell to suit requirements.

**SV\_CARETBLINKTIME** \*  
Caret blink rate, in milliseconds

**SV\_DBLCLKTIME** \*  
Mouse doubleclick time, in milliseconds

**SV\_CXDBLCLK** \*  
Width of mouse doubleclick sensitive area

**SV\_CYDBLCLK** \*  
Height of mouse doubleclick sensitive area

**SV\_SBREPEATTIME** \*  
Scroll bar auto-repeat interval time, in milliseconds

**SV\_ALARMFREQ** \*  
The frequency of the alarm signal generated by a call to **WinBeep()** if its **UINT** parameter is **0xffff**.

**SV\_ALARMDURATION** \*  
The duration of the alarm signal generated by a call to **WinBeep()** if its **UINT** parameter is **0xffff**.

**SV\_ARROWCURSOR**  
Arrow cursor handle

SV\_IBEAMCURSOR  
Text I-Beam cursor handle

SV\_HOURLASSCURSOR  
Hourglass cursor handle

SV\_UPARROWCURSOR  
Up-arrow cursor handle

SV\_SIZECURSOR  
Size cursor handle

SV\_MOVECURSOR  
Move cursor handle

### 4.1.25 Using Windows of Other Threads

If an application is using a window handle of another thread or process, it is possible that the thread that owns the window might destroy the window, thereby invalidating the window handle. Even worse, after the window is destroyed, another thread may create a new window, that has the same window handle as the one previously destroyed. In this case, the application with the window handle of the invalid window handle now has a valid window handle of a completely different window.

In order to prevent this problem, it is possible to "Lock" a window. A locked window cannot be destroyed; `WinDestroyWindow()` waits until the window is unlocked before proceeding. During this time, messages may be received from other applications, that may possibly involve the window being destroyed.

Most functions that return a window handle have the option of returning that window handle locked or unlocked. If the function is being used in a context where it is guaranteed that the returned window handle is not of another thread, or the window handle is simply used for equality checking against `NULL` or a window of the current thread, it is not necessary to lock the window handle. If the window handle is used as an argument to other functions, or is sent messages, then the window handle must be locked.

It is very important that a locked window handle be unlocked at some point. Otherwise, this may hang the application that owns the window when it attempts to destroy the window.

To help guard against the possibility of an application leaving windows locked, the `WinCheckWindowLockCount()` function can be called to check to see if any windows remain locked by the application. This function is typically called in an application's main loop. This test is also made automatically when the application is terminated.

Windows should not be locked for indeterminate lengths of time. In order to use a window of another application for a long time, in order to store the handle in a global variable, etc., the window can be "Destroy Registered". When a destroy registered window is destroyed, all top level windows in the system are notified with a `WM_OTHERWINDOWDESTROYED` message that the window has been destroyed. This message is sent before the window is actually destroyed. The idea is that the saved window handle may be invalidated by the application that saved the window handle. Before using a destroy registered window as an argument to a function or `WinSendMsg()`, it should be locked as usual.

#### 4.1.25.1 Window Locking Functions

---

##### WinLockWindow

---

###### Format

```
HWND WinLockWindow (hwnd)
HWND hwnd;
```

###### Description

This function locks the specified window, preventing it from being destroyed. This function is used in conjunction with `WinUnlockWindow` to ensure that a window handle of another application does not get destroyed while an application is using it.

If `WinDestroyWindow()` is called with a locked window handle, it is not actually destroyed until the window is unlocked.

`WinLockWindow()` returns `hwnd` if successful, or `NULL` if the window has been destroyed or is otherwise invalid.

The window is marked as being locked by incrementing a lock count. Therefore, the same window may be locked more than once by different applications. The number of `WinUnlockWindow()` calls must match `WinLockWindow()` calls.

---

##### WinUnlockWindow

---

###### Format

```
UINT WinUnlockWindow (hwnd)
HWND hwnd;
```

Description

This routine unlocks a window. This is accomplished by decrementing the lock count that was incremented by `WinLockWindow()`. The window is not actually unlocked until this count reaches 0.

Returns the lock count of the window after the unlock is performed. If the window was actually unlocked, 0 is returned.

---

`WinGetWindowLockCount`

---

Format

```
INT WinGetWindowLockCount (hwnd)
HWND hwnd;
```

Description

This function returns the lock count of the specified window, or 0 if the window is not locked.

Since a window may be locked by another thread or process at any time, the value returned by this function may also change at any time.

---

`WinCheckWindowLockCount`

---

Format

```
VOID WinCheckWindowLockCount (hab, cLock)
HAB hab;
INT cLock;
```

Description

In debugging versions of the system, this routine will produce an error message on a debugging terminal if the number of windows locked but not unlocked by the current thread is not equal to `cLock`. This function can be used to help ensure that applications don't leave other application's windows locked.

In non-debug versions of the system, `WinCheckWindowLockCount` has no effect.



## 4.1.26 Window Destroy Registration

---

### WinRegisterWindowDestroy

---

#### Format

```
BOOL WinRegisterWindowDestroy(HWND hwnd, BOOL fRegister)
HWND hwnd;
BOOL fRegister;
```

#### Description

This function provides a mechanism whereby an application will be notified when a window of another thread is destroyed.

If `fRegister` is `TRUE`, this function registers the given window so that when it is destroyed, a `WM_OTHERWINDOWDESTROYED` message is broadcast to all top level windows of other tasks.

Registering the window is accomplished by incrementing a register count. If `fRegister` is `FALSE`, this routine unregisters the window by decrementing the register count. The window is not actually unregistered until the count reaches 0.

---

### WM\_OTHERWINDOWDESTROYED

---

#### Format

```
WM_OTHERWINDOWDESTROYED
lParam1:  HWND hwndDestroyed;
lParam2:  OL
Returns:  OL
```

#### Description

This message is sent to all top-level windows when a window registered with `WinRegisterWindowDestroy()` is destroyed. `lParam1` contains the window handle of the window being destroyed. The message is sent by `WinDestroyWindow()` after the window has been hidden, but before the window is actually destroyed.

This message is non-queued.

## 4.1.27 System and Queue Hooks

Presentation Manager provides a mechanism by which procedures written by an application programmer can get called when interesting things happen in the system. Such procedures are called *Hooks*. These hooks allow things such as filtering of mouse & keyboard input before an application receives it.

There are a number of places in the system where Hooks can be installed - ie. there are a number of different types of interesting events that can be hooked out and processed in some special ways. More than one procedure can be installed for a given type of event. In this case, the procedures are 'chained' together so that each event is first passed to one procedure and then to the next, and so on down the chain. This chain of procedures is called a *Hook Chain*.

Within a particular Hook Chain, the procedures are called in the order that they were installed. If a hook procedure returns FALSE, the next hook in the chain is called. If a hook procedure returns TRUE, then the next hook in the chain is not called.

There are two kinds of hook chains: the System Chain and the Queue Chain. The system chain applies to all running Presentation Manager applications, but the queue chain applies only to the thread associated with the queue that has the hook installed. In this way, it is possible to install hooks that affect the entire system or just a particular thread.

Procedures on a system chain (system hooks) may be called in the process/thread context of any running Presentation Manager application. Procedures on a queue chain (queue hooks) are called only in the context of the process/thread associated with the hooked queue. This has consequences for the way in which hook procedures are defined.

System Hook procedures *must* be defined in library modules because it is not possible to call application module procedures from other applications. However, it is possible for an application to install one of its own procedures as a queue hook into one of its own input queues without defining the procedure in a library module.

---

### WinSetHook

---

#### Format

```
BOOL WinSetHook(hab, hmq, iHook, lpfnHook,
                idModule)
HAB hab;
HMQ hmq;
INT iHook;
FARPROC lpfnHook;
UINT idModule;
```

**Description**

Installs the lpfnHook procedure into the hook chain specified by iHook and hmq. idModule, returned by the DOS DosLoadModule call, is the module handle that contains the hook procedure. If idModule is NULL, the hook procedure is in the current module. iHook is a HK\_\* value from the table below that specifies which hook is being installed. Returns TRUE if successful, FALSE otherwise.

If hmq is NULL, then the hook is installed in the system hook chain; otherwise, the hook is installed in the specified queue's hook chain. For a given hook, queue hooks are always called before system hooks. System hooks are always added to the end of the chain, and queue hooks are added to the beginning of the chain.

**Notes**

Use WinQueryWindowULong() to obtain the queue handle associated with a window handle.

The value HMQ\_CURRENT can be used as the hmq parameter to indicate the current queue.

---

**WinReleaseHook**

---

**Format**

```
BOOL WinReleaseHook(hab, hmq, iHook, lpfnHook,
                    idModule)
HAB hab;
HMQ hmq;
INT iHook;
FARPROC lpfnHook;
UINT idModule;
```

**Description**

Unhooks the specified hook procedure from the hook chain specified by iHook and hmq. Returns TRUE if successful, FALSE otherwise.

If hmq is NULL, then the hook is unhooked from the system hook chain; otherwise, the hook is removed from the specified queue's hook chain.

The value HMQ\_CURRENT can be used as the hmq parameter to indicate the current queue.

#### 4.1.27.1 Hook example

```
InitCode()
{
    ...
    WinSetHook(NULL,
               HK_EXAMPLE,
               WinHookProc);
    ...
}

BOOL FAR PASCAL ExampleHookProc(hc, lParam1, lParam2)
INT hc;
ULONG lParam1;
ULONG lParam2;
{
    switch (hc) {
    case 0:
        if (fProcessHook) {
            ExampleProcess(lParam1, lParam2);
            /*
             * Return TRUE to indicate that we've processed
             * the call, which will prevent the next guy on
             * the chain from being called.
             */
            return(TRUE);
        }
        break;
    ...
    }
    /*
     * Return FALSE to indicate that we have not processed the
     * call to cause the message to be passed on to the next
     * link in the hook chain.
     */
    return(FALSE);
}
```

#### 4.1.27.2 Queue Hook codes

Here is a list of the available hook codes. Complete documentation for each hook can be found elsewhere in this manual.

---

##### Window Hook Codes

###### HK\_INPUT

Input hook: called when message is removed from app queue before msg is returned by WinGetMsg() or WinPeekMsg()

###### HK\_MSGFILTER

Msg filter hook: called inside system mode loops

**HK\_SENDMSG**

WinSendMsg hook: called as a message is being sent

The following hooks may be installed only in the system hook chain:

---

**HK\_JOURNALRECORD**

Journal recording hook

**HK\_JOURNALPLAYBACK**

Journal playback hook

**4.1.27.3 HK\_INPUT hook**

The HK\_INPUT hook is called when messages are removed from an application queue, before being returned by WinGetMsg() or WinPeekMsg().

---

**InputHook**

---

**Format**

```
BOOL FAR PASCAL InputHook(lpQmsg)
LPQMSG lpQmsg;
```

**Description**

lpQmsg is a pointer to a QMSG structure. This hook is called from within get/peekmsg just before returning to the application with the message that will be returned. There are no restrictions on calls that may be made at this time.

If this hook returns TRUE, the message is not passed on to the application. If it returns FALSE, then the message is passed on.

**4.1.27.4 HK\_MSGFILTER hook**

The HK\_MSGFILTER hook is called inside any of the system mode loops, such as during size/move tracking, while a dialog box or menu is up, etc.

---

**MsgFilterHook**

---

**Format**

```
BOOL FAR PASCAL MsgFilterHook( msgf, lpQmsg)
INT msgf;
LPQMSG lpQmsg;
```

**Description**

lpQmsg is a pointer to a QMSG structure, and msgf is a code indicating the context that the hook procedure is called in. The msgf code may be one of the following values:

---

Message Filter Context Codes  
Context code

MSGF\_DIALOGBOX  
DialogBox() mode loop

MSGF\_MESSAGEBOX  
MsgBox() mode loop

MSGF\_MENU  
Menu tracking

MSGF\_MOVE  
Window movement tracking

MSGF\_SIZE  
Window size tracking

MSGF\_SCROLLBAR  
Scroll bar tracking

MSGF\_NEXTWINDOW  
Window enumeration mode loop

If this hook procedure returns TRUE, the message has been processed by the hook and will not be processed by the mode loop code. If it returns false, the message will be processed by the mode loop code.

---

**CallMsgFilter**

---

**Format**

```
BOOL CallMsgFilter(lpQmsg, msgf)
LPQMSG lpQmsg;
INT msgf;
```

**Description**

This function allows the application to call the HK\_MSGFILTER hook procedure. msgf may be one of the standard MSGF\_\* values, or an application-specific code. WinCallMsgFilter() returns TRUE if any of the message filter hooks returns TRUE, FALSE if they all return FALSE.

#### 4.1.27.5 HK\_SENDMSG hook

```
typedef struct {  
    ULONG lParam2;  
    ULONG lParam1;  
    UINT msg;  
    HWND hwnd;  
} SMHSTRUCT;
```

---

##### SendMsgHook

---

###### Format

```
BOOL FAR PASCAL SendMsgHook(lpsmh, fInterTask)  
BOOL fInterTask;  
SMHSTRUCT FAR *lpsmh;
```

###### Description

The WinSendMsg hook is called whenever a window procedure is called via WinSendMsg() or WinDispatchMsg. fInterTask is TRUE if the message is an inter-task WinSendMsg, and FALSE if intra-task. lpsmh is a pointer to a SMHSTRUCT structure, defined above.

The fields of the SMHSTRUCT contain the WinSendMsg parameters. There is no special return value for the WinSendMsg hook.

#### 4.1.27.6 HK\_HELP hook

---

##### HelpHook

---

###### Format

```
BOOL FAR PASCAL HelpHook(context, idTopic,  
                          idSubTopic, lprcPosition)  
UINT context;  
UINT idTopic;  
UINT idSubTopic;  
LPRECT lprcPosition;
```

###### Description

This hook can be called directly by an application or in the default processing associated with windows, menus and message boxes:

- WinDefWindowProc calls the HK\_HELP hook with idTopic == window ID of window sent the message, idSubTopic == window ID of window with focus or 0xffff if no window has the focus, and

lprcPosition == window rect (in screen coordinates) of window with the focus or window sent the message if idSubTopic == 0xffff.

- Menu code: calls the HK\_HELP hook with idTopic == window ID of currently selected submenu, and idSubTopic == menu item ID of currently selected submenu item, or 0xffff if no item is selected. lprcPosition is the bounding rectangle of the selected item, or of the top level menu if idSubTopic == 0xffff.
- Message Box code if WM\_HELP message is received while in a message box, then the help hook is called with idTopic == message box ID and idSubTopic == id of button, which is the same as message box return value corresponding to that button. lprcPosition is the bounding rectangle of the button.

#### 4.1.27.7 HK\_JOURNALRECORD and HK\_JOURNALPLAYBACK hooks

---

##### JournalRecordHook

---

###### Format

```
BOOL FAR PASCAL JournalRecordHook(lpqmsg)
QMSG FAR *lpqmsg;
```

###### Description

lpqmsg points to a QMSG structure, which contains the message to be recorded.

This hook is called AFTER raw input has been translated to WM\_CHAR or WM\_?BUTTONDOWNCLK messages. lpqmsg->hwnd is also setup when the hook is called.

---

##### JournalPlaybackHook

---

###### Format

```
BOOL FAR PASCAL JournalPlaybackHook(lpqmsg, fSkip)
QMSG FAR *lpqmsg;
BOOL      fSkip;
```

###### Description

lpqmsg points to a QMSG structure, where the message to be played back is to be returned.



When `fSkip` is `FALSE`, the journal playback hook returns the next message available. The same message should be returned each time, until it is skipped with a call with `fSkip == TRUE` (see below).

The time at which the message will be ready to be played back should be returned in `lpqmsg->time`. This time value may be greater than the current value. When this hook is called, the `lpqmsg->time` field is initialized to the current time, which can be used to determine whether the next message is ready or not. This value, rather than the result of `WinGetCurrentTime()`, should be used for any delta calculations performed by the hook procedure.

If `fSkip` is `TRUE`, then the journal playback hook should skip to the next message. The `lpqmsg` parameter is `NULL` in this case.

---

## WM\_QUEUESTATUS

---

### Format

```
WM_QUEUESTATUS
lParam1:    queue status (result of WinQueryQueueStatus())
lParam2:    OL
Returns:    OL
```

### Description

The recording hook gets called by call to `WinGetQueueStatus` if the results of `WinGetQueueStatus()` changed since the last call.

The playback hook should return these messages synchronized with everything else.

**Notes** This message is only used for journal recording and playback hooks.

### 4.1.27.7.1 Notes on journal recording and playback

- Although the record hook is called after raw input has been translated, it is necessary only to record the raw input data. For `WM_CHAR` msgs, only the `KC_` flags, `lpqmsg->time`, and the high order byte of `lpqmsg->lParam1` (where the scan code is stored). In this case, `KC_CHAR` should be 0. For mouse input messages, double click messages may be recorded as simply single click messages.
- On playback, it is not necessary to specify the `lpqmsg->hwnd` or `lpqmsg->pt` fields: these fields will always be filled in by the input code.

- Conversely, it is possible to play back only virtual keys and character codes, by not setting the `KC_SCANCODE` bit. In this case, applications will receive `WM_CHAR` messages that do not have the `KC_SCANCODE` bit set.
- The journal recording hook is called after message translation occurs, but before the key is translated via the system or queue accelerator tables.
- With double click messages, a problem can arise when messages are played back with the double click time different than when the messages were recorded. The problem is that single clicks during recording may be interpreted as double clicks, or vice versa. Applications that make use of these hooks should probably save & restore the state of the system timing variables: double click time, cursor flash rate, repeat, etc., to ensure that no timing problems will arise in this fashion.
- `WinGetMsg/WinPeekMsg()` may not be called inside either the journal record or playback hook.

## 4.1.28 International Information

### 4.1.28.1 Overview

The functions described here deal with problems associated with different keyboards and code pages.

Input translation depends upon the physical keyboard, and, for characters, on the code page in use.

There are also functions which allow the application to translate between any two supported code pages.

A number of national language facilities are provided by base DOS. Further ones described here supplement these.

### 4.1.28.2 Input Translation

#### *4.1.28.2.1 Concepts*

Keyboard input is obtained in the form of messages received via `WinGetMsg`.

A `WM_CHAR` message is delivered for each keydown and keyup for all keys on the keyboard. Translation of the scan code received from the keyboard is done by the `WinGetMsg` call that receives the keystroke.

WinGetMsg does not send a message for every typematic repeat from the keyboard; it may buffer typematic repeats into one or more messages. Each message contains a count, which is the number of typematic repeats since the first keydown, or since the last WinGetMsg call. This count will begin at one for the first keydown.

Keyboard data along with mouse data is buffered asynchronously into the Presentation Manager system queue. Keyboard data is removed from the system queue when the application that owns the input focus calls WinGetMsg or WinPeekMsg. Only one keyboard event is dequeued at a time.

Translation occurs when the event is dequeued. The message obtained from WinGetMsg contains three separate fields that represent the key pressed: the hardware dependent scan code, the virtual key code and the codepoint or dead key. These are discussed below:

- The virtual key (VKEY) concept is that there should be a virtual key code for each "word" (eg esc or F1) on the keytops of the keyboard. Consistency requires that most applications should use the PC set of virtual keys built in to Presentation Manager, but applications with special requirements can define their own virtual key sets. An example of valid use of this capability is mainframe applications accessed via a terminal emulator. These use words such as clear, PA1, etc in contrast to PC applications which use esc, home etc.
- The code point (CKEY) concept is that there should be a code point value for every key on the keyboard with a symbol on it. The code points can be either ASCII or EBCDIC and are country dependent. Effectively, each code point corresponds to a unique "glyph" that can appear on the screen.
- Some keys with words on them (eg Enter) generate both a virtual key value and a code point. This is because the key does need to be treated as a function key in some applications, but also has a defined ASCII code point associated with it which some applications may prefer to use.
- For CKEY values that correspond to dead keys (e.g. umlaut) WinGetMsg will identify these CKEYs with a special flag in the WM\_CHAR message. It is the application's responsibility to echo the dead key in the appropriate manner (i.e. without advancing the cursor). If the next CKEY after the dead key is a valid dead key combination, then another flag will be set in the WM\_CHAR message to identify the composite character. Again it is the application's responsibility to echo the character appropriately. There are three situations the application must deal with:
  - valid dead key combination should replace the dead key display with the new composite character

- invalid dead key combination (except the space character) should leave the dead key displayed, advance the cursor and display the new CKEY, followed by a beep.
- dead key followed by a space should leave the dead key displayed and advance the cursor.
- The valid set of dead keys and their combinations with other keys is defined for each supported code page.

#### *4.1.28.2.2 Keystroke Translation*

Presentation Manager keystroke translation provides full flexibility in remapping, support of country specific keyboard layouts, and support of EBCDIC and ASCII code pages. This is achieved by the use of three types of table which are described below:

- The key to VKEY table (VKeyXLateTbl). This table generates virtual key codes based on the key pressed and the shift state. Presentation Manager supplies two tables of this type covering the PC VKEY set for the two physically different keyboards.
- The key to Universal Glyph List (UGL) table (GlyphXLateTbl). (The UGL is a list of all (non-DBCS) glyphs that can be generated by a Presentation Manager application using standard Presentation Manager facilities. All glyphs for all supported languages, plus the APL glyphs, are included in the UGL.) The key to UGL table is always used in conjunction with the UGL to CKEY table described below. It is uniquely defined by the layout of the keytops. Presentation Manager supplies tables of this type corresponding to all supported keyboard layouts.
- The UGL to CKEY table (CharXLateTbl). This table is used in conjunction with the previous one. Presentation Manager supplies a table of this type for each supported code page. Also included in this table is the dead key table, which defines the valid dead keys for each code page and the valid dead key combinations.
- The WinPeekMsg and WinGetMsg API calls control the translation process that generates the virtual key and character code values that are in the WM\_CHAR message. The translation process consists of the following steps:
  - Apply scan code/keyboard state to VKeyXLateTbl. Result is a virtual key.
  - Apply scan code/keyboard state to GlyphXLateTbl. Result is a glyph code.
  - Apply the glyph code to CharXLateTbl. Result is a character code, with appropriate dead key bits set.

- The translation tables that control the above process are determined at boot time, based on the physical keyboard type and the values specified in CONFIG.SYS.

#### 4.1.28.3 String/Character Translation

The functions in this section translate between any two of the code pages supported.

---

##### WinCpTranslateString

---

###### Format

```
BOOL WinCpTranslateString (hab, cpSrc, lpSrc,
                           cpDest, lenDest, lpDest)
HAB hab;
LONG cpSrc;
LPSTR lpSrc;
LONG cpDest;
LONG lenDest;
LPSTR lpDest;
```

###### Description

Translates a string from one code page to another. Both source and translated strings are null-terminated.

The source string buffer pointed to by lpSrc is not altered. The translated string is written to the buffer pointed to by lpDest, up to a maximum of lenDest bytes.

###### Returns

---

**FALSE** Error; possible errors are:

- Dissimilar code pages. No translation is possible.
- Neither code page is recognized.
- The source code page is not recognized.
- The destination code page is not recognized.
- The translated string is too long. Truncation has occurred.

**TRUE** Successful translation of most if not all characters. Character substitution will occur for all untranslated characters.

---

**WinCpTranslateChar**

---

**Format**

```
UCHAR WinCpTranslateChar (hab, cpSrc, chSrc,  
                           cpDest)  
HAB hab;  
LONG cpSrc;  
UCHAR chSrc;  
LONG cpDest;
```

**Description**

Translates a character from one code page to another.

---

**Returns**

---

- FALSE** Error; possible errors are:
- Dissimilar code pages. No translation is possible.
  - Neither code page is recognized.
  - The source code page is not recognized.
  - The destination code page is not recognized.
  - Translation requires more than one byte.
- TRUE** The translated character in the destination code page. If an exact match was not possible for this character between the specified code pages, the character will have been translated to a standard character.

**4.1.28.4 String functions**

This section describes the functions dealing with strings that depends on the current international informations set for the application.

To order strings a two dimensionnal table must be used. Each rows consists of characters (eg: 'A', 'A' umlaut, 'a', 'a' umlaut, etc...). The characters are ordered within a row (secondary order), but this ordering is secondary to the order of the row themselves (primary order). Here is an example of what could be a section of this table:

```
...  
Ä, A acute, A grave, A umlaut, a, a acute, a grave, a umlaut  
B, b  
C, C cedilla, c, c cedilla  
D, d  
E, E acute, E grave, E umlaut, e, e acute, e grave, e umlaut
```

...

This mechanism is not only useful for accentuated strings ordering but also for case sensitive ordering. For example:

```
'A' < 'a',  
'Bc' < 'bc', but  
'Bc' > 'ba'
```

In the third example the primary ordering of the 2nd characters ('c' and 'a') takes precedence over the secondary ordering of the 1st characters ('A' and 'a'). In fact the secondary ordering is only considered when the 2 strings are found equal following the primary ordering rule. By else when comparing strings with different lengths the characters in the longer string that don't have their counterpart are considered greater and the secondary ordering is ignored. For example:

```
'a' < 'ab' and  
'a' < 'Ab'.
```

---

## WinCompareStrings

---

### Format

```
int WinCompareStrings(idcp, lpsz1, lpsz2, fSecondary)  
UINT idcp;  
LPSTR lpsz1;  
LPSTR lpsz2;  
BOOL fSecondary;
```

### Description

This function compares the two strings pointed to by lpsz1 and lpsz2, it takes into consideration primary and secondary order if fSecondary is TRUE, and only primary order if fSecondary is FALSE.

It returns:

```
0   if strings are equal,  
1   if string pointed to by lpsz1 is < string  
    pointed to by lpsz2  
-1  if string pointed to by lpsz1 is > string  
    pointed to by lpsz2
```

---

## WinIsAlpha

---

### Format

```
BOOL WinIsAlpha(idcp, lpsz)  
UINT idcp;  
LPSTR lpsz;
```

### Description

This functions checks if the string pointed to by lpsz is only made of alphabetical character. It returns TRUE if this is the case, FALSE otherwise.

Typically are valid any alphabetical characters with accents or marks like cedilla. Characters like space, comma, accents, etc... are not valid.

---

## WinLower

---

### Format

```
UINT WinLower(idcp, lpsz)
UINT idcp;
LPSTR lpsz;
```

### Description

This function converts the given string to lower case. lpsz is a long pointer to a null-terminated string, that is updated in-place.

The return value is the length of the converted string.

---

## WinUpper

---

### Format

```
UINT WinUpper(idcp, lpsz)
UINT idcp;
LPSTR lpsz;
```

### Description

This function converts a string or a character to upper case. lpsz is a long pointer to a null-terminated string, that is updated in-place.

The return value is the length of the converted string.

---

## WinUppe

---

### Format

```
UINT WinUpperChar(idcp, wInchar);
UINT idcp;
UINT wInchar;
```

### Description

This function converts a character in wInchar to upper case.

The return value is the converted character, or zero if



an invalid character was detected.

---

### WinLowerChar

---

#### Format

```
UINT WinLowerChar(idcp, wInchar);  
UINT idcp;  
UINT wInchar;
```

#### Description

This function converts a character in wInchar to lower case.

The return value is the converted character, or zero if an invalid character was detected.

---

### WinNextChar

---

#### Format

```
LPSTR WinNextChar(idcp, lpCurrentChar)  
UINT idcp;  
LPSTR lpCurrentChar;
```

#### Description

This function moves to the next character in a string. lpCurrentChar is a long pointer to a character in a null terminated string.

The return value is a long pointer to the next character in the string, or if there is no next character, to the null character at the end of the string.

#### Notes

CPNext is used to move through strings whose characters are two or more bytes each (for example, strings containing characters from a Japanese character set).

---

### WinPrevChar

---

#### Format

```
LPSTR WinPrevChar(idcp, lpStart, lpCurrentChar)  
UINT idcp;  
LPSTR lpStart;  
LPSTR lpCurrentChar;
```

#### Description

This function moves to the previous character in a string. lpStart is a long pointer to the beginning of the string. lpCurrentChar is a long pointer to a character in a null terminated string.

The return value is a long pointer to the previous character in the string, or to the first character in the string if lpCurrentChar is equal to lpStart.

Notes WinPrevChar() is used to move through strings whose characters are two or more bytes each (for example, strings containing characters from a Japanese character set).

#### 4.1.28.5 Code Pages Available

---

##### WinQueryCpList

---

###### Format

```
LONG WinQueryCpList (hab, count, array)
HAB hab;
LONG count;
LONG array[];
```

###### Description

This returns a list of the code pages currently available. A maximum of count code pages is returned in array.

###### Returns

---

```
FA SE Error
TRUE Number of code pages returned
```

#### 4.1.29 Miscellaneous

---

##### WM\_SYSTEMERROR

---

###### Format

```
WM_SYSTEMERROR
lParam1: 0
lParam2: 0L
Returns: 0L
```

###### Description

This message is sent to all top-level windows when an out of memory system error occurs during WinInitialize() or WinCreateMsgQueue(). The shell window notifies the user of the out of memory condition.

If the window receiving the message is the frame

window, it repeats the message to the client window, the window with the window id of `STDID_CLIENT`. See "Window Frames".

(

(

(

# Index

---

BM\_CLICK message, 279  
BM\_QUERYCHECK message, 283  
BM\_QUERYCHECKINDEX message,  
    280  
BM\_QUERYHILITE message, 282  
BM\_SETCHECK message, 282  
BM\_SETHILITE message, 282  
BOOL type, 41  
Bottom Window, 163  
  
CallMsgFilter, 364  
CARETINFO type, 318  
Child Window, 163  
CLASSINFO type, 196  
clipboard viewer, 89  
closing a task, 84  
CONFIG.SYS, 88  
control panel, 86  
Control Panel, 137  
CREATESTRUCT type, 170  
CS\_SAVEBITS, 177  
CS\_SYNCPAINT, 215  
CURSORINFO type, 322  
  
Data Structures  
    gis, 110, 145  
    parameter descriptor block, 114, 117,  
        120, 147  
    Program Handle array, 146  
    Program Information Block, 113, 116,  
        119, 146  
    ProgramEntry, 111, 145, 148  
    Switch List Block, 134, 150  
    switch list control block, 125, 128,  
        134, 149, 150, 152  
    Switch List entry definition, 134,  
        150, 152  
    xywinsize, 112, 133, 153  
Destroy Registered Windows, 357  
Dialog Control Groups, 252  
Dialog Procedure, 247  
Dialog Template, 247  
DialogProc, 248  
direct manipulation, 69  
directory tree, 63

EM\_CLEAR message, 288  
EM\_COPY message, 288  
EM\_CUT message, 288  
EM\_GETCHANGED message, 286  
EM\_PASTE message, 288  
EM\_QUERYSEL message, 286  
EM\_SETFONT message, 287  
EM\_SETSEL message, 286  
EM\_SETTEXTLIMIT message, 287  
ending a task, 84  
extended selection., 53

FARPROC type, 41  
FID\_MINMAX, 246  
FID\_TITLEBAR, 243  
FID\_WIDEST, 245  
File Cabinet, 61  
files, direct manipulation, 69  
focus, input, 58

Group Information Structure, 110, 145

HANDLE type, 41  
HCURSOR type, 321  
help, 91  
    additional notes, 95  
HelpHook, 365  
help  
    invoking, 92  
    shell help index, 94  
    the help window, 92

initial view, 91  
initialization file, 91  
input focus, 58  
InputHook, 363  
installation, 107

JournalPlaybackHook, 366  
JournalRecordHook, 366

keyboard input, 58  
keyboard  
    key actions, 54

keyboard (*continued*)  
 selecting items, 52

LM\_DELETEITEM message, 293  
 LM\_GETSELECTION message, 294  
 LM\_INSERTITEM message, 292  
 LM\_QUERYITEMCOUNT message,  
 291  
 LM\_QUERYITEMHANDLE message,  
 296  
 LM\_QUERYITEMTEXT message, 295  
 LM\_QUERYITEMTEXTLENGTH  
 message, 295  
 LM\_QUERYTOPINDEX message, 293  
 LM\_SEARCHSTRING message, 296  
 LM\_SELECTITEM message, 293  
 LM\_SETITEMHANDLE message, 296  
 LM\_SETITEMTEXT message, 294  
 LM\_SETTOPINDEX message, 292  
 Locked Windows, 356  
 LONG type, 41  
 LPRECT type, 337  
 LPSTR type, 41

MENUITEM type, 307  
 MM\_DELETEITEM message, 313  
 MM\_ENDMENUMODE message, 312  
 MM\_GETSELITEMID message, 314  
 MM\_INSERTITEM message, 312  
 MM\_ITEMIDFROMPOSITION  
 message, 317  
 MM\_ITEMPOSITIONFROMID  
 message, 316  
 MM\_QUERYITEM message, 314  
 MM\_QUERYITEMATTR message,  
 317  
 MM\_QUERYITEMCOUNT message,  
 317  
 MM\_QUERYITEMTEXT message, 315  
 MM\_QUERYITEMTEXTLENGTH  
 message, 315  
 MM\_REMOVEITEM message, 313  
 MM\_SELECTITEM message, 313  
 MM\_SETITEM message, 316  
 MM\_SETITEMATTR message, 318  
 MM\_SETITEMHANDLE message, 315  
 MM\_SETITEMTEXT message, 316  
 MM\_STARTMENUMODE message,  
 311  
 Mnemonics., 306  
 mouse  
   button actions, 56  
   pointer, 51  
   selecting items, 52

MsgFilterHook, 363  
 multiple selection, 53

Object Windows, 165  
 online help, 91  
 Orphaned Windows, 165  
 Owned Window, 165

parameter descriptor block, 114, 117,  
 120, 147  
 Parent Window, 163  
 Parking-Lot, 60  
 POINT type, 337  
 pointer, 51  
 Private Window Class, 161  
 Program Handle array, 146  
 Program Information Block, 113, 116,  
 119, 146  
 program titles, 110  
 ProgramEntry, 111, 145, 148  
 programs, starting with the Shell, 75  
 Public Window Class, 161

RECT type, 337  
 restore size and position, 61

SBML\_QUERYPOS message, 301  
 SBML\_QUERYRANGE message, 301  
 SBML\_SETPOS message, 300  
 SBML\_SETSCROLLBAR message, 300  
 selecting items, 52  
 selection cursor, 51  
 SendMsgHook, 365  
 SHANDLE type, 41  
 shell  
   general features, 50  
   help index, 94  
 Sibling Window, 163  
 SMHSTRUCT type, 365  
 starting OS/2 PM, 91  
 starting programs, 75  
 STARTUP, 75, 76  
 STARTUP editor, 77  
 stopping a task, 84  
 Subclassing of Windows, 166  
 Switch List Block, 134, 150  
 switch list control block, 125, 128, 134,  
 149, 150, 152  
 Switch List entry definition, 134, 150,  
 152  
 SWP type, 188  
 Sync Paint Window, 215

System Menu, 58  
 SZM\_TRACKSIZE message, 246

Task Manager, 81  
 TBM\_QUERYSTATE message, 243  
 TBM\_SETSTATE message, 244  
 TBM\_TRACKMOVE message, 244  
 terminating a task, 84  
 titles, of programs, 110  
 Top level windows, 163  
 Top Window, 163  
 tree window, 63

UCHAR type, 41  
 UINT type, 41  
 ULONG type, 41  
 Update Region, 214

WC\_BUTTON, 278  
 WC\_EDIT, 284  
 WC\_LISTBOX, 289  
 WC\_MINMAX, 246  
 WC\_SIZE, 245  
 WC\_TITLEBAR, 243  
 WinAlarm, 264  
 WinBeginEnumWindows, 198  
 WinBeginPaint, 218  
 WinBeginPaint function, 215  
 WinCalcFrameRect, 240  
 WinCheckWindowLockCount, 358  
 WinCloseClipbrd, 328  
 WinCompareStrings, 373  
 WinCopyAccelTable, 346  
 WinCopyRect, 339  
 WinCpTranslateChar, 372  
 WinCpTranslateString, 371  
 WinCreateAccelTable, 345  
 WinCreateCaret, 318  
 WinCreateCursor, 322  
 WinCreateDlg, 255  
 WinCreateFrameControls, 240  
 WinCreateStdWindow, 238  
 WinCreateStdWindowIndirect, 239  
 WinCreateWindow, 171  
 WinDestroyAccelTable, 346  
 WinDestroyCaret, 320  
 WinDestroyCursor, 323  
 WinDestroyWindow, 173  
 WinDismissDlg, 257  
 WinDlgBox, 256  
 window appearance, 57  
 Window Coordinates, 162  
 Window Descendants, 163

Window Invalidation, 214  
 Window Owner, 165  
 Window Validation, 214  
 windows  
   changing size, 60  
   controlling, 57  
   maximizing, 59  
   minimizing, 59  
   moving, 61  
   restoring size and position, 61  
   using the System Menu, 58  
 WinDrawBorder, 228  
 WinDrawIcon, 228  
 WinDrawText, 226  
 WinEmptyClipbrd, 328  
 WinEnableWindow, 175  
 WinEnableWindowUpdate, 178  
 WinEndEnumWindows, 198  
 WinEndPaint, 218  
 WinEndPaint function, 215  
 WinEnumClipbrdFmts, 330  
 WinEnumDlgItem, 260  
 WinEnumWindow, 198  
 WinEqualRect, 339  
 WinExcludeUpdateRgn, 222  
 WinFlashWindow, 207  
 WinFormatFrame, 241  
 WinGetActiveWindow, 204  
 WinGetCursorPos, 324  
 WinGetPS, 216  
 WinGetSysModalWindow, 208  
 WinGetSysValue, 352  
 WinGetWindowLockCount, 358  
 WinHalftoneBitmap, 227  
 WinInflateRect, 340  
 WinIntersectRect, 341  
 WinInvalidateRect, 219  
 WinInvalidateRgn, 219  
 WinInvertRect, 229  
 WinIsAlpha, 373  
 WinIsChild, 186  
 WinIsRectEmpty, 338  
 WinIsThreadActive, 207  
 WinIsWindow, 185  
 WinIsWindowEnabled, 176  
 WinIsWindowVisible, 179  
 WinLoadAccelTable, 345  
 WinLoadCursor, 322  
 WinLoadDlg, 254  
 WinLoadString, 343  
 WinLockScreen, 223  
 WinLockVisRgns, 224  
 WinLockWindow, 357  
 WinLower, 374  
 WinLowerChar, 375  
 WinMakeGPoint, 342

- WinMakeGRect, 342
- WinMapDlgPoints, 259
- WinMapWindowPoints, 200
- WinMessageBox, 261
- WinMultWindowFromIDs, 184
- WinNextChar, 375
- WinOffsetRect, 340
- WinOpenClipbrd, 328
- WinOpenWindowDC, 217
- WinPrevChar, 375
- WinProcessDlg, 255
- WinProcessDlgMsg, 259
- WinPtInRect, 340
- WinQueryAccelTable, 348
- WinQueryCaretInfo, 321
- WinQueryClassInfo, 196
- WinQueryClassName, 170
- WinQueryClipbrdData, 330
- WinQueryClipbrdFmtInfo, 331
- WinQueryClipbrdOwner, 329
- WinQueryClipbrdViewer, 335
- WinQueryCpList, 376
- WinQueryCursorInfo, 325
- WinQueryDesktopWindow, 182
- WinQueryDlgItemInt, 258
- WinQuerySysColor, 351
- WinQueryUpdateRect, 220
- WinQueryUpdateRgn, 221
- WinQueryWindow, 185
- WinQueryWindowParams, 181
- WinQueryWindowProcess, 186
- WinQueryWindowRect, 186
- WinQueryWindowText, 179
- WinQueryWindowTextLength, 180
- WinQueryWindowUInt, 201
- WinQueryWindowULong, 202
- WinRegisterClass, 169
- WinRegisterWindowDestroy, 359
- WinReleaseHook, 361
- WinReleasePS, 217
- WinRestrictCursor, 325
- WinScrollWindow, 224
- WinSendDlgItemMsg, 257
- WinSetAccelTable, 347
- WinSetActiveWindow, 203
- WinSetClipbrdData, 329
- WinSetClipbrdOwner, 329
- WinSetClipbrdViewer, 334
- WinSetCursor, 323
- WinSetCursorPos, 324
- WinSetDlgItemInt, 258
- WinSetHook, 360
- WinSetMultWindowPos, 191
- WinSetOwner, 188
- WinSetParent, 187
- WinSetRect, 338
- WinSetRectEmpty, 339
- WinSetSysColors, 351
- WinSetSysModalWindow, 208
- WinSetSysValue, 353
- WinSetWindowParams, 181
- WinSetWindowPos, 188
- WinSetWindowText, 180
- WinSetWindowUInt, 202
- WinSetWindowULong, 202
- WinShowCaret, 320
- WinShowCursor, 323
- WinShowWindow, 177
- WinSubclassWindow, 196
- WinSubstituteStrings, 264
- WinSubtractRect, 341
- WinTranslateAccel, 346
- WinUnionRect, 341
- WinUnlockWindow, 357
- WinUpdateWindow, 221
- WinUpper, 374
- WinUpperChar, 374
- WinValidateRect, 220
- WinValidateRgn, 220
- WinWindowFromID, 184
- WinWindowFromPoint, 199
- WM\_ACTIVATE message, 205
- WM\_ACTIVATETHREAD message, 206
- WM\_ADJUSTWINDOWRECT message, 270
- WM\_CALCVVALIDRECTS message, 193
- WM\_COMMAND message, 273
- WM\_CONTROL: Button Notification message, 281
- WM\_CONTROL message, 273
- WM\_CONTROLCURSOR message, 275
- WM\_CONTROLHEAP message, 275
- WM\_CREATE message, 174
- WM\_DESTROY message, 175
- WM\_DESTROYCLIPBOARD message, 332
- WM\_DRAWCLIPBOARD message, 335
- WM\_DRAWITEM message, 290
- WM\_ENABLE message, 176
- WM\_ERASEBACKGROUND message, 234
- WM\_FORMATFRAME message, 233
- WM\_HELP message, 274
- WM\_HSCROLL message, 298
- WM\_HSCROLLCLIPBOARD message, 334
- WM\_INITDIALOG message, 253
- WM\_MEASUREITEM message, 291



WM\_MOVE message, 193  
 WM\_OTHERWINDOWDESTROYED  
     message, 359  
 WM\_PAINT message, 215, 223  
 WM\_PAINTCLIPBOARD message,  
     332  
 WM\_QUERYACCELTABLE message,  
     348  
 WM\_QUERYDLGCODE message, 272  
 WM\_QUERYMINMAXINFO message,  
     237  
 WM\_QUERYMOVESIZEINFO  
     message, 245  
 WM\_QUEYTWINDOWPARAMS  
     message, 183  
 WM\_QUEUSTATUS, 367  
 WM\_RENDERALLFMTS message,  
     332  
 WM\_RENDERFMT message, 331  
 WM\_SETACCELTABLE message, 348  
 WM\_SETITEMHEIGHT message, 295  
 WM\_SETWINDOWPARAMS  
     message, 183  
 WM\_SHOW message, 179  
 WM\_SIZE message, 192  
 WM\_SIZECLIPBOARD message, 333  
 WM\_SUBSTITUTESTRING message,  
     265  
 WM\_SYSCOLORCHANGE message,  
     352  
 WM\_SYSTEMERROR message, 376  
 WM\_UPDATEFRAME message, 234  
 WM\_VALIDATEACCEL message, 349  
 WM\_VSCROLL message, 298  
 WM\_VSCROLLCLIPBOARD message,  
     334

xywinsize, 112, 133, 153

Z-ordering, 59

1

2

3